

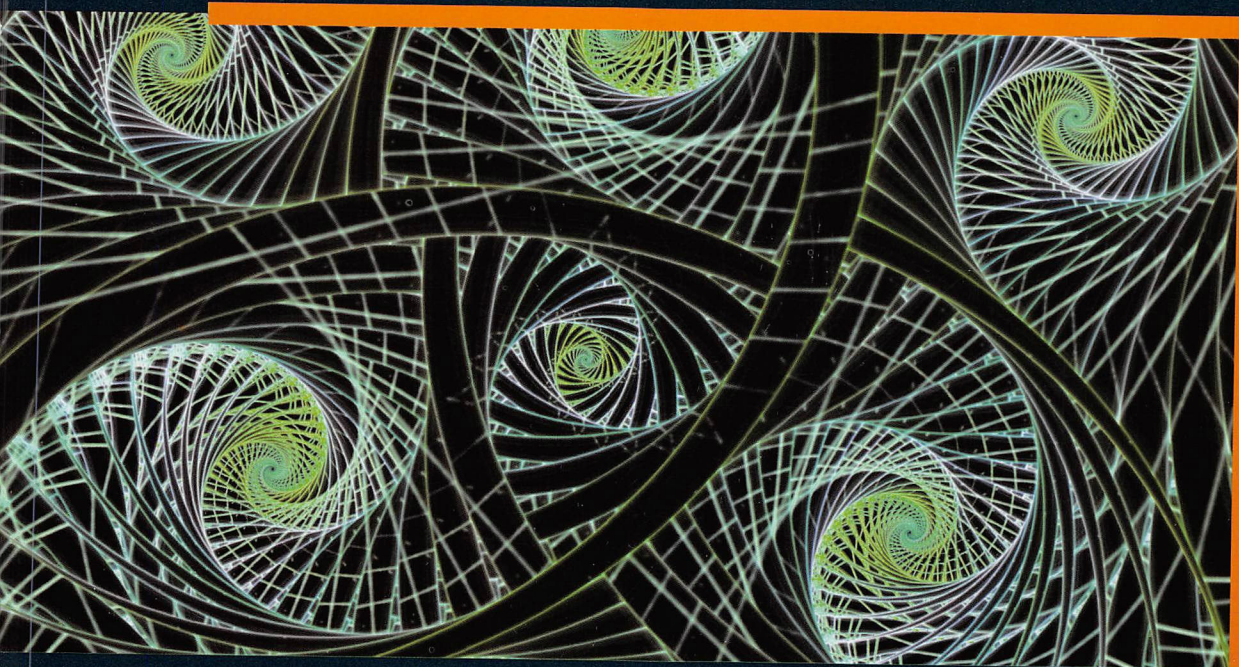
版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



Julia语言程序设计

Introduction to Julia Programming



零基础学习Julia语言，基于版本v1.0
系统全面地介绍Julia编程语言，从基本概念到编程要点，从多维数组到并行计算，包含大量示例代码，以及机器学习综合案例

魏坤 编著



机械工业出版社
China Machine Press





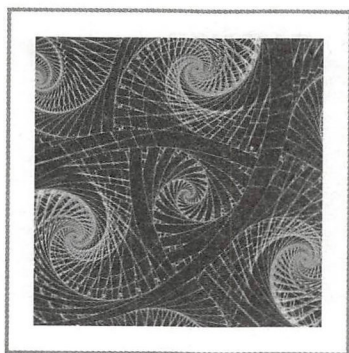
内容简介

本书系统介绍Julia编程语言的基本概念、各种功能、编程要点，包含大量示例代码以及编程技巧。全书共17章。第1~2章介绍Julia语言的基本特点、基础概念。第3~4章介绍Julia语言的数值系统、各种运算符使用规则。第5~6章介绍类型系统，以及经典的判断、循环逻辑。第7~8章介绍函数与方法，以及Julia的核心优势——多维数组。第9~11章介绍字符串与日期处理方法以及元编程。第12章介绍与IO相关的内容，包括流、文件操作、网络通信及序列化等。第13章介绍Julia代码的组织方式，包括模块、文件以及包。第14章介绍Julia原生提供的并行计算特性，是Julia中颇具魅力的内容之一。第15章介绍Julia与C/C++、Python进行混合编程的基本方法。第16章总结了Julia编程方面的经验以及优化建议。第17章给出了编程案例。本书内容丰富，讲解细腻，适合于所有软件开发人员，以及高等院校相关专业师生。





华章程序员书库



Julia语言程序设计

魏坤 编著



机械工业出版社
China Machine Press





图书在版编目 (CIP) 数据

Julia 语言程序设计 / 魏坤编著. —北京: 机械工业出版社, 2018.10
(华章程序员书库)

ISBN 978-7-111-60757-1

I. J… II. 魏… III. 程序设计—研究 IV. TP312

中国版本图书馆 CIP 数据核字 (2018) 第 195441 号

Julia 语言程序设计

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 冯秀泳

责任校对: 李秋荣

印 刷: 北京市兆成印刷有限责任公司

版 次: 2018 年 10 月第 1 版第 1 次印刷

开 本: 186mm × 240mm 1/16

印 张: 27.5

书 号: ISBN 978-7-111-60757-1

定 价: 99.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东





Preface 前言

几十年前，科学家为了避免处理重复、单调的事情，比如反复地按一套公式计算结果等，发明了计算机。计算机其实是工业自动化的一个产物，可以说是工业化时代的巅峰代表。当时人们怎么也不会想到，计算机的发展带来了互联网，而互联网导致了信息化时代的到来。如今，在数据蔓延、不断渗透的过程中，智能化代表了未来的发展方向。

与此同时，数据的不断累积、膨胀、延伸引发了计算领域的深刻变化，而且数据的密集性和分布性也提出了大量的计算密集性和分布式要求。很多工业级的生产场景中，在要求开发高效率、维护低成本、运行高可靠的同时，还需要具备高性能的特点。Julia 语言应运而生。

为何撰写本书

Julia 借助于 JIT 动态解析器及其优秀的设计机制，在一些计算特性上能达到静态语言的性能，这是非常令人惊讶的，也是吸引笔者的地方。笔者在大数据挖掘与机器学习领域浸淫十几年，面对种类繁多、数量巨大、计算逻辑复杂的各种问题，深感性能与开发效率极为重要。本想浅尝，但却再无法舍弃，Julia 语言的各种特性令笔者兴奋不已。

几年前我开始接触 Julia，其自然快捷的编写感受，顺畅舒适的体验，与现在广为使用的 Python 颇为相近。但 Julia 更多的是以科学与数值计算为目的，原生的并发机制与分布式、云计算特性，简洁人性化的语法，以及媲美于静态语言的性能，所有这些表现都是笔者期待已久的。而今，在日常的数据分析和前期数据处理中，笔者都会首选 Julia 语言。

为了让喜爱的 Julia 语言能够更快普及，进入首选的工业级技术架构，笔者不揣浅陋，捉笔从文，写就此书，以求与各位爱好者共同进步。也期冀 Julia 能成为一个写着简单、读着愉悦、迁移方便、应用广泛、性能强劲的通用编程语言，让我们在开发工作中不再纠结语言的选择。

Julia 的官方文档还算详细，但组织结构并不清晰，概念散乱各处，对初学者并不友好。为此，笔者愿意以此书为契机，将这几年的经验分享给大家，希望能更条理清晰地展





现 Julia 的特色，帮助大家更快、更好地熟悉并掌握 Julia，并在实际开发中获益。

本书的结构

Julia 语言不仅提供了灵活、多样、简洁的语法，更有很多符合实际开发需求的强大特性，也充满了人性化的设计。它不仅支持各种类型的声明定义、贴近于数学概念的计算规则，还在常见的高维数组、字符串处理、国际化支持、元编程等方面提供了强大的支持。尤其是在并行计算、混合编程等方面更是独具特色，原生地提供了良好的机制，使得这方面的编程工作变得极为快捷便利。

为了能够让读者通过本书了解、认识、掌握 Julia 语言的基本概念并能付诸实践，笔者反复对掌握的资料进行了梳理、调整，并且基于真实的运行环境，尽可能地为每个功能点提供相应的示例代码，以求准确、明晰地阐明各个要点。本书主要内容如下：

第 1 章介绍 Julia 语言的基本情况，同时重点介绍 Julia 运行环境的使用方法。

第 2 章对编程语言的基础概念进行了简单的介绍，能够帮助读者在后续的学习中理解 Julia 语言的特点，对于有经验的读者可做选读内容。

第 3 章从包括有理数、复数在内的基本数值系统开始，详细地介绍 Julia 语言的基本语法。

第 4 章基于前一章介绍的各种数值类型介绍 Julia 的各种运算符使用规则。

第 5 章主要介绍经典的判断、循环逻辑，还有 Julia 中较为特别的复合表达式。

第 6 章介绍类型系统，这是 Julia 语言的精髓，包括抽象类型、元类型及复合类型等，都有着 Julia 自己的特点。另外，该章还会重点介绍类型参数化的内容，这也是 Julia 灵活适应各种应用场景的基础。该章介绍的元组、字典、集合等结构也是开发 Julia 程序时经常会用到的数集。

第 7 章介绍函数与方法，这不但是 Julia 多态分发机制的基础，也是 Julia 博采众长的精华。

第 8 章介绍 Julia 被称为数值计算语言的核心优势特性——多维数组。数组是科学计算中最为常见的数据结构，但能够以统一的结构表达向量、矩阵、张量甚至高维空间的机制，却是 Julia 的特色。

第 9 章介绍开发中经常遇到的字符串处理方法，包括常见的正则表达式等。

第 10 章使我们能够更深刻地认识 Julia 中“一切皆对象”的理念，因为通过 Symbol 与 Expr 类型的封装，Julia 代码也是对象的一部分。这章介绍的宏，也是在 Julia 开发中极为强大的特性。

第 11 章介绍时间和日期的处理方法。

第 12 章介绍与 IO 相关的内容，包括流、文件操作、网络通信及序列化等。通过该章的学习，我们会再一次为 Julia 的简洁、高效所折服。

第 13 章介绍 Julia 代码的组织方式，包括模块、文件以及包，尤其是对包的管理进行了较为详尽的阐述。





第 14 章介绍 Julia 原生提供的并行计算特性，这也是 Julia 最具魅力的内容之一。在该章中，我们会详尽地阐述协程任务、远程调用及引用，还有数据通道等方面的内容。

第 15 章可以作为选读内容，介绍 Julia 与 C/C++、Python 进行混合编程的基本方法。不过由于运行环境等方面的约束，在学习该章时，如果要通过实例进行实践，建议在 Linux 或 MacOS 下进行。

第 16 章给出了对 Julia 编程方面的经验总结或优化建议。该章也可作为选读内容，不过笔者仍建议所有的读者能认真学习该章内容，并通过实例进行体验，这样才能对 Julia 语言有更为深刻的认识。

第 17 章以机器学习领域中经典的决策树算法为例，展示如何用 Julia 实现该算法的主要过程。在这个实践中，我们对 Julia 各种语法技巧的使用会有更切实的认知。

在本书的结尾，以附录的方式列举了 Julia 中常见的异常类型、系统常量以及字符串操作函数，而且对可能有用的第三方包进行了简单的介绍，希望读者能够通过这方面的内容，了解 Julia 社区的强大力量，习惯性地从社区中获得各种支持。

另外，在本书撰写时，为了简明扼要地将概念阐述清楚，在确保不会影响读者了解语言的核心应用要点的情况下，在内容上进行了适当缩减。如果读者想了解更深入的内容，可以通过官网查阅更多的资料进行学习。

本书适合的读者

本书尽量从基础知识入手，逐步深入地介绍 Julia 语言。但因为 Julia 语言的设计与实现借鉴了众多先进的理念，所以本书难以进行大而全的阐述，省略了不少内容。所以本书不适合没有任何编程经验的读者，读者至少要了解面向对象、泛型编程与函数式等编程概念。

由于本书几乎涉及了 Julia 语言的方方面面，要点颇多，所以建议读者在通过本书学习 Julia 语言时，能够按照其中的实例，多多动手实践，并能在实际的编程工作中选用 Julia 语言，进行一些开发实践。无论哪一种语言，动手实践是掌握这门语言的唯一捷径。

致谢

首先感谢设计与实现 Julia 语言的近 700 位贡献者，为计算机与科学领域提供了一门简洁易用的语言；也感谢近 2000 个第三方库的社区贡献者，让 Julia 语言能够快速普及，焕发出了蓬勃的生机。

此外，感谢上海交通大学副教授潘汉博士在本书校对期间给予的大力支持。

在本书数月的撰写过程中，妻子冯莹霞和家人的支持与照顾让笔者感动不已，有了她们本书才能够有机会顺利完成，与读者们相见。感谢她们给予我的一切！



目 录 Contents

前言

第 1 章 初识 Julia 1

- 1.1 有用的资源 2
- 1.2 环境准备 4
 - 1.2.1 二进制包安装 5
 - 1.2.2 编译安装 7
- 1.3 交互式控制台 8
- 1.4 命名规则与关键字 11
- 1.5 先睹为快 12
 - 1.5.1 Hello World 12
 - 1.5.2 体型分布案例 13
 - 1.5.3 小结 18

第 2 章 基础概念 19

- 2.1 静态与动态语言 19
- 2.2 内存管理 20
- 2.3 经典编程范式 21

第 3 章 数值系统 24

- 3.1 整型 24
 - 3.1.1 表达方式 25

3.1.2 类型强制限定 27

3.1.3 有无符号转换 28

3.2 布尔型 29

3.3 浮点型 30

3.3.1 基本定义 31

3.3.2 零的表达 32

3.3.3 epsilon 34

3.3.4 无穷值 35

3.3.5 非数值 37

3.3.6 内置常量 37

3.4 有理数型 38

3.5 复数型 40

3.6 随机数 42

3.7 任意精度算术 43

第 4 章 运算符 46

4.1 算术运算符 46

4.2 位运算符 51

4.3 更新运算符 55

4.4 比较运算符 56

4.5 逻辑运算符 60

4.6 运算优先级	61	6.8.2 参数化抽象类型	102
4.7 类型提升	62	6.8.3 参数化元类型	105
第5章 控制逻辑	64	6.8.4 参数化基本原理	106
5.1 复合表达式	64	6.8.5 参数化继承关系	108
5.2 判断逻辑	65	6.8.6 协变与逆变	110
5.3 循环逻辑	67	6.9 常用数集	112
5.3.1 while	67	6.9.1 元组	113
5.3.2 for	69	6.9.2 键值对	115
5.4 异常处理	73	6.9.3 字典	117
5.4.1 异常触发	73	6.9.4 集合	121
5.4.2 异常捕捉	74	6.10 缺失值的表达	123
第6章 类型系统	77	6.10.1 missing	123
6.1 类型简介	77	6.10.2 nothing	125
6.2 抽象类型	78	6.10.3 可有可无的表达	125
6.3 元类型	80	第7章 函数	129
6.4 类型操作	83	7.1 基本定义	129
6.4.1 弱类型机制	83	7.1.1 常规结构	129
6.4.2 类型断言	84	7.1.2 类型限定	130
6.4.3 Data Type	85	7.1.3 共享传参	132
6.4.4 类型别称	86	7.1.4 数集展开式调用	133
6.4.5 继承关系	87	7.1.5 多返回值	134
6.5 复合类型	88	7.2 参数传递方式	134
6.5.1 基本定义	88	7.2.1 默认参数	134
6.5.2 默认构造函数	90	7.2.2 键值参数	135
6.5.3 成员访问及不可变性	91	7.2.3 可变参数	137
6.5.4 单例复合类型	93	7.3 函数对象	140
6.6 类型联合	94	7.3.1 Function 类型	140
6.7 TypeVar	96	7.3.2 函数作为参数	141
6.8 类型参数化	97	7.3.3 函数作为返回值	143
6.8.1 参数化复合类型	97	7.4 匿名函数	144



VIII

7.5	参数化方法	146
7.6	多态分发	148
7.7	复合类型构造方法	153
7.7.1	外部构造方法	153
7.7.2	内部构造方法	155

第8章 多维数组 158

8.1	创建数组	158
8.1.1	串联方式	160
8.1.2	辅助构造函数	163
8.1.3	范围表达式	164
8.1.4	推导式	168
8.2	索引访问	169
8.3	遍历迭代	176
8.4	子数组与视图	179
8.4.1	范围切片	179
8.4.2	逻辑索引	180
8.4.3	局部视图	180
8.5	稀疏数组	182
8.5.1	典型稀疏结构	183
8.5.2	结构转换	184
8.5.3	内容映射	186
8.6	矢量化计算	189
8.6.1	map 函数	189
8.6.2	广播	192
8.6.3	点操作	193
8.6.4	数组运算符	196
8.7	排序	197
8.8	查找	200
8.9	missing 作为元素	205
8.10	线性代数中的矩阵处理	207

8.10.1	矩阵操作	207
8.10.2	特殊矩阵	208
8.10.3	矩阵分解	211

第9章 字符串 217

9.1	字符	217
9.2	String 对象	220
9.2.1	表达	220
9.2.2	索引	221
9.2.3	遍历	223
9.2.4	子串	224
9.3	变量替换	225
9.4	正则表达式	226
9.5	常用操作	229
9.5.1	连接	229
9.5.2	比较	232
9.5.3	搜索	232
9.5.4	替换	234
9.5.5	分割	235
9.6	字节数组	237
9.7	与数值的转换	239

第10章 元编程 241

10.1	Symbol 类型	241
10.2	Expr 类型	243
10.2.1	构造	243
10.2.2	衍生	248
10.3	宏	249
10.3.1	定义	250
10.3.2	调用	250
10.3.3	预定义宏	251

第 11 章 时间与日期	255	13.4 包	309
11.1 类型	255	13.4.1 管理机制	309
11.2 构造	257	13.4.2 安装移除	312
11.3 访问	260	13.4.3 更新固化	317
11.4 解析	262	13.4.4 小结	318
11.5 运算	265	第 14 章 并行计算	319
11.5.1 早晚比较	265	14.1 基础概念	319
11.5.2 时长计算	267	14.1.1 进程与线程	319
11.5.3 时间序列	269	14.1.2 条件变量	320
11.5.4 周期舍入	270	14.2 协程调度	321
11.6 属性	273	14.3 数据通道	325
第 12 章 流与 IO	275	14.3.1 Channel 对象	325
12.1 标准流	275	14.3.2 通道绑定	330
12.2 文件操作	278	14.4 远程调用与远程引用	332
12.3 读写缓存	281	14.5 共享数组	345
12.4 流的回溯	284	14.6 方法小结	348
12.5 序列化	287	第 15 章 混合编程	351
12.6 网络通信	290	15.1 运行外部程序	351
第 13 章 组织结构	294	15.2 调用 C/C++	352
13.1 模块	294	15.2.1 链接库操作	352
13.1.1 基本定义	294	15.2.2 函数调用	353
13.1.2 标准模块	296	15.2.3 数据访问	356
13.1.3 模块路径	298	15.2.4 C++ 接口	358
13.1.4 预编译	298	15.3 嵌入 C/C++	358
13.2 模块与脚本文件	299	15.4 与 Python 互调	362
13.3 变量域	300	第 16 章 Julia 编程规范	364
13.3.1 全局域	302	16.1 文档注释	364
13.3.2 局部域	302		
13.3.3 let 关键字	305		

16.2 高性能编程建议	368	第 17 章 编程实战	389
16.2.1 类型	369	17.1 决策树基本概念	389
16.2.2 函数	373	17.2 决策树分类器的实现	391
16.2.3 数组	377	17.3 随机森林算法的构建	406
16.2.4 IO	381	附录 A 内置异常类型	409
16.2.5 其他	381	附录 B 内置系统常量	411
16.3 与其他语言的异同	382	附录 C 字符串操作函数	413
16.3.1 与 Python 相比	382	附录 D 常用包简介	416
16.3.2 与 Matlab 相比	384	后记	428
16.3.3 与 R 相比	385		
16.4 Julia 代码风格	387		

初识 Julia

Julia 语言^①是一种为高性能数值计算设计的高层次动态编程语言，在分布式并行化、精确数值计算等方面提供了独具特色的支持，并包含大量可扩展的数学函数库。尤其是在线性代数、随机数生成、信号处理、字符串处理等方面，Julia 集成了众多成熟、优秀的基于 C 和 Fortran 开发的开源库，有着很高的性能与效率。另外，Julia 的开发者社区已经非常强大，贡献了大量的第三方库，我们可通过内置的包管理器方便地安装使用。

Julia 语言更多的特点还有：

- 多态分发（Multiple Dispatch）机制，通过不同类型的参数组合，可以定义同名函数不同的行为。
- 动态类型系统：用户自定义的类型可像内置类型一样快速、轻便。
- 简洁又可扩展的数值类型转换与提升机制。
- 高效能的多语言编码环境，支持包括 UTF-8 在内的各种 Unicode 编码^②。
- 原生设计的并行与分布式计算机制。
- 轻量级的“绿色”线程——协程机制。
- 优秀的性能，可以与静态编译的 C 语言媲美。
- Lisp 语言式的宏及元编程（Meta-programming）范式的支持。
- 内置的第三方功能包管理器。
- 可与 Python、R、Matlab 及 Java 等语言进行混合编程。
- 类似于 Shell 的外部程序调用。

① Julia 语言官网为 <https://julialang.org>，其中包括 Julia 基本介绍、源代码链接、英文说明文档、博客、社区、生态及教程等各种资源。

② Unicode 是计算机领域的一个业界标准，统一为各种自然语言字符设定了唯一的编码，以满足跨语言、跨平台文本转换处理的需求。UTF-8、UTF-16、UTF-32 都是编码方案之一。

□ 不需要额外的封装层或特别的 API，即可直接调用 C 语言的库函数。

可以说 Julia 在很多方面都独具特色。比如在并行化计算方面，Julia 并没有专门设计特殊的语法结构，而是提供了足够灵活的机制，并可自动进行分布式的部署，能够实现云端操作，使得并行化编程极为便捷。

值得称道的是，Julia 语言基于 MIT 许可证^①是开源、免费的。而且其生态中的各种库与软件也主要采用 GPL、LGPL 或 BSD 等许可。核心代码及各种第三方大部分均托管在 GitHub 这个有名的开源代码管理平台中，用户可以获得源代码，了解语言的各种实现细节，不但能对语言进行更深入的学习，也能够在设计思路方面受益。

使用 Julia 开发有着非常好的体验。不但语法自然简洁，而且结构清晰，效率也非常高。更为可贵的是，其性能也不差。优秀的语言设计结合强大的即时（Just-In-Time, JIT）编译系统 LLVM^②，使得 Julia 的运行性能在很多时候能够媲美 C 语言。在一份官方提供的 Benchmark 中，相比于 C、Fortran、Python、Matlab/Octave、R、JavaScript、Java、Lua 与 Mathematica 等其他语言，Julia 在性能方面有着非常卓越的表现。^③

Julia 语言 v0.1 版发布于 2013 年 2 月 14 日，自 2017 年 12 月 14 日发布 v0.6.2 版本后，代码更新很快，2018 年 8 月 8 日发布了 v0.7 版后，当年 8 月 9 日便发布了 v1.0 的正式版本。经过数次的版本迭代，Julia 语言已经日趋成熟，并已经科学计算领域崭露头角。完全可以相信，在不久的将来，在众多的编程语言中，尤其是在科学计算领域方面，Julia 必能占有一席之地。

1.1 有用的资源

Julia 语言的设计者们显然是聪明的一群人，但笔者认为他们更具有符合时代的智慧。他们在设计实现这门包罗万象又简洁高效的语言时，便建立了开放的包管理机制，从而能够借助强大的开源社区，让 Julia 以前所未有的速度发展与普及。

截至本书成稿时，官方注册的包已近 2000 个，已经成为 Julia 生态系统的重要组成部分。笔者相信，随着 Julia 的快速发展，第三方包的规模与质量也会不断地提升。

Julia 的贡献者来自世界各地，提供了大量各自领域有针对性的包，在这些包中要找到我们想要的支持并不困难。为了更好地管理这些包，让这些包更好地服务于各种应用场景，Julia 将这些包分成多个频道，列举如下：

-
- ① MIT 许可证（The MIT License）是开源软件授权中被广泛使用的一种，比 GPL、LGPL、BSD 等更为宽松：被授权人有权使用、复制、修改、合并、出版、传播、再授权及販售软件或其副本，也可适当修改授权条款，但需在发布版本中包含该版权许可声明。
 - ② 底层虚拟机（Low Level Virtual Machine, LLVM），是编译器的基础建设之一，利用虚拟技术实现编译期、链接期、运行期及“空闲期”的优化，是一系列技术工具链的集合，通常作为某类语言的编译器的后台。LLVM 项目由伊利诺伊大学的维克拉姆·艾夫（Vikram Adve）和克里斯·拉特纳（Chris Lattner）于 2000 年发起。早期以 C/C++ 为主要对象，后来开始支持 Objective-C，并扩展到其他更多的编程语言。
 - ③ 有兴趣的读者可以从 GitHub 中的官网地址下载代码运行看看，网址为 <https://github.com/JuliaLang/Microbenchmarks.git>。

通用类：

- JuliaDocs——Julia 文档系统相关的包。
- Julia-i18n——国际化 (i18n) 与本地化 (L10n) 支持。
- JuliaTime——日期与时间相关的库。
- JuliaPraxis——最佳实践案例与支持。
- JuliaEditorSupport——文本编辑器与 IDE 的扩展及插件。
- Juno——基于 Atom 编辑器的 Juno IDE。

基础计算：

- JuliaArrays——自定义的数组类型及相关工具。
- JuliaBerry——Raspberry Pi[⊖]相关的资源与支持组件。
- JuliaCI——用于 Julia 包的持续集成工具。
- JuliaGPU——GPU 计算支持。
- JuliaInterop——与其他语言进行混合编程的相关支持包。
- JuliaIO——包括序列化、通信协议及文件格式等 IO 相关的包。
- JuliaParallel——并行与分布式计算支持。
- JuliaWeb——Web 技术栈

数学：

- JuliaDiff——微分数值计算。
- JuliaDiffEq——微分方程求解与分析。
- JuliaGeometry——计算几何。
- JuliaGraphs——图理论与实现。
- JuliaIntervals——计算机精准算术支持。
- JuliaMath——包括积分、傅里叶变换、插值等在内的数学包。
- JuliaOpt——最优化。
- JuliaPolyhedra——多面体计算 (polyhedral computation)。
- JuliaSparse——稀疏矩阵求解等支持。

科学：

- BioJulia——生物学。
- EcoJulia——生态学。
- JuliaAstro——天文学。
- JuliaDSP——数字信号处理。
- JuliaQuant——金融。
- JuliaQuantum——量子科学与技术。

⊖ 树莓派基金会 (Raspberry Pi) 是英国一个小型的慈善组织, 旨在推广科技, 且不以技术销售营利为目的。同名的 Raspberry Pi 则是该基金会提供的一款针对电脑业余爱好者、教师、小学生以及小型企业等用户的迷你电脑, 预装 Linux 系统, 体积仅信用卡大小, 搭载 ARM 架构处理器, 运算性能和智能手机相仿。

- ❑ JuliaPhysics——物理学。
- ❑ JuliaDynamics——线性及非线性动态系统、混沌等。

数据：

- ❑ JuliaML——机器学习。
- ❑ JuliaStats——数理与统计。
- ❑ JuliaImages——图像处理。
- ❑ JuliaText——自然语言处理、计算语言学及信息检索。
- ❑ JuliaDatabases——数据库及数据仓库驱动支持。
- ❑ JuliaData——数据操纵、存取及 IO 相关。

可视化：

- ❑ GiovineItalia——图表支持。
- ❑ JuliaPlots——数据可视化。
- ❑ JuliaGL——OpenGL API 及其生态。
- ❑ JuliaGraphics——绘图、色彩及 GUI 相关支持。

在官网中，我们可以通过这些频道快速地找到感兴趣的内容。另外，Julia 社区提供了论坛、年会等线下活动，也提供 Twitter、新闻组等线上方式，帮助开发者参与并了解 Julia 语言，并能够促进参与者的交流互动。而且在 YouTube 等平台提供了各种教程，帮助新学者熟悉 Julia 语言。

官方提供的开发文档、GitHub 中开源的代码，还有耗费心思撰写的本书，都可以成为我们学习这门语言的起点。

1.2 环境准备

Julia 语言是跨平台的，主要支持 Windows、Linux 及 MacOS 三类操作系统，不但能够在 i686 及 x86_64 架构上运行，而且还支持 ARM、PowerPC 等平台。所支持的环境详细如表 1-1 所示。

表 1-1 Julia 支持的环境

操作系统	架构	持续集成 (CI)	官方二进制安装程序	支持层面
Linux 2.6.18+	x86-64 (64 位)	✓	✓	官方
	i686 (32 位)	✓	✓	官方
	ARM v7 (32 位)		✓	官方
	ARM v8 (64 位)		✓	官方
	PowerPC (64 位)			社区
	PTX (64 位)	✓		第三方
macOS 10.8+	x86-64 (64 位)	✓	✓	官方

(续)

操作系统	架构	持续集成 (CI)	官方二进制安装程序	支持层面
Windows 7+	x86-64 (64 位)	✓	✓	官方
	i686 (32 位)	✓	✓	官方
FreeBSD 11.0+	x86-64 (64 位)	✓		社区

我们可以从其官方网站中直接下载编译好的二进制安装包，也可以下载源代码在本机的环境中重新编译。不过建议使用提供的可执行程序安装 Julia 程序环境，除非非常熟悉 C++ 代码的编译过程。

1.2.1 二进制包安装

Julia 官方为 Windows、MacOS、Linux 及 FreeBSD 几个最常见的平台提供了已经编译好的可直接执行的安装程序，可通过官方页面 <https://julialang.org/downloads> 进行下载。安装文件并不大，30~80MB，不像微软各种东西，动辄就上 GB。二进制的安装方式能让我们不用操心各种依赖与复杂的编译过程，安装完毕即可轻松开始使用。

1. Windows

Julia 支持 Windows 7/ Windows Server 2012 及更新的操作系统，所以最好不要在 Windows 2000 或 Windows XP 这些太老的系统中尝试，不确定会发生什么问题。32 位版本的 Julia 可运行在 32 位 (x86) 或 64 位 (x86_64) 的操作系统中，但 64 位版本只能运行在 64 位的 Windows 系统中。

在 Windows 上安装 Julia 极为简单，将 exe 安装包下载后，双击文件然后按照引导一步步执行既可。需要说明的是，对于 Windows 7 或 Windows Server 2012 系统，需要额外进行以下两个工作：

1) 更新操作系统组件，以支持 1.1 及 1.2 版的传输层安全协议 (Transport Layer Security, TLS)。这是因为 GitHub 已经停止了 TLS 1.0 的使用，所以需要操作系统支持更高的 TLS 版本，Julia 的包管理器才能正常工作。

2) 安装 Windows Management Framework (WMF) 3.0 版或者更高版本。

若要卸载 Julia，只需直接将 Julia 的安装目录以及工作目录（一般为 %HOME%/.julia 路径）直接删除，并将 %HOME%/.juliarc.jl 和 %HOME%/.julia_history 两个文件同时删除即可。

2. MacOS

Julia 支持 MacOS 10.8 及更新的发行版本，提供的二进制安装包是一个扩展名为 dmg 的可执行程序，包含了 Julia.app 目录，可以直接打开执行。

除了二进制安装包，MacOS 还可以使用 HomeBrew 安装 Julia 环境，执行语句如下：

```
$ brew update
```



```
$ brew tap staticfloat/julia
$ brew install julia
```

或者采用 cask 的方式，即：

```
$ brew cask install julia
```

在安装完成后，在系统终端中执行

```
$ julia
```

即可启动 Julia 环境。

如果要卸载 Julia，只需移除 Julia.app 目录及 ~/.julia 工作目录，如果不再需要配置文件，可将 ~/.juliarc.jl 文件同时删除即可。

3. Linux 及 FreeBSD

除了 FreeBSD 外，Julia 还支持多种 Linux 发行版本，包括 Fedora、RHEL、CentOS、Scientific Linux、Oracle Enterprise Linux、Ubuntu、Debian、openSUSE 及 Arch Linux 等。

下载官方通用的 tar.gz 格式二进制包后，安装并没有特别的执行步骤，解压后便可执行。为了后续的方便，最好是将执行目录添加到系统环境变量中，简单的办法是在 /usr/local/bin 或 /usr/bin 目录中建立安装目录中 bin/julia 可执行文件的软链接 (Symbolic Link)，语句一般为：

```
$ sudo ln -s <where you extracted the julia archive>/bin/julia /usr/local/bin/julia
```

此后便可无须切换到安装目录或输入完整的路径来执行 Julia 主程序了。

实际上，对于 Linux 及 FreeBSD 来说，通过其中的包管理命令安装 Julia 可能更为方便，而且也能够更便捷地对 Julia 进行更新。

对于使用 RHEL、CentOS、Scientific Linux 及 Oracle Enterprise Linux (版本 5 或更高) 的用户来说，需要开启操作系统的 EPEL (Extra Packages for Enterprise Linux) 支持，然后便可与 Fedora (版本 19 或更高) 一样，执行下述安装命令：

```
$ sudo dnf copr enable nalimilan/julia
$ sudo yum install julia
```

若使用的是 CentOS 7 或更高版本，也可直接执行：

```
$ sudo yum-config-manager --add-repo https://copr.fedorainfracloud.org/coprs/
    nalimilan/julia/repo/epel-7 /nalimilan-julia-epel-7.repo
$ sudo yum install julia
```

若发行版中没有 dnf 与 yum-config-manager，需在 Copr 网站中下载相关的 .repo 文件，复制到系统的 /etc/yum.repos 源管理目录，再次尝试执行安装过程。

每日构建 (Nightly Building) 是一些系统或程序通常采用的更新方式，如果需要及时跟踪 Julia 的最新功能，可以通过 yum 对已经安装的 Julia 程序进行更新。首先，执行：

```
$ sudo dnf copr enable nalimilan/julia-nightlies
```

以开启每日更新版本库，再执行以下命令时：

```
$ sudo yum install julia
```

安装的 Julia 将是 Nightly 的最新版本。之后，如果需要更新版本，执行以下命令即可：

```
$ sudo yum upgrade julia
```

不过，由于每日构建版本一般是开发过程的每日快照，有些功能后期仍存在修正调整的可能，测试也并不充分，容易存在各种 Bug，所以对于普通的使用者来说，使用 Nightly 版本时还需要谨慎。

对于 Ubuntu 发行版，采用的是 apt 包管理器，在发布的最新版本中可通过 apt-get 命令（或新发行版中的 apt 命令）直接进行安装，相关的包文件包括 cantor-backend-julia、julia-common、julia-dbg、julia-doc 及 libjulia-dev、libjulia 等。如果要及时获得 Julia 的最新开发版本，可以添加 Nightly 的源，命令如下：

```
$ sudo add-apt-repository ppa:staticfloat/julianightlies
$ sudo add-apt-repository ppa:staticfloat/julia-deps
$ sudo apt-get update
```

然后执行安装程序或在需要时执行升级命令。

在 FreeBSD 系统中同样简单，可通过 Ports Collection 直接安装 Julia 程序，执行的安装命令为：

```
pkg install julia
```

如果不再需要 Linux 或 FreeBSD 中已经安装的 Julia 程序，卸载的方式取决于安装的方式。通过 apt-get/apt 或 yum 安装时，执行 apt-get remove julia 或 yum remove julia 即可。如果是二进制下载的方式，移除解压的目录并删除 ~/.julia 目录及 ~/.juliarc.jl 配置文件。

1.2.2 编译安装

如果想通过源码编译的方式安装 Julia 程序，在确定这么做之前，我们先了解一下 Julia 代码所依赖的第三方库，列举如下：

- ❑ LLVM (6.0) 编译器基础库。
- ❑ FemtoLisp 编译器前端。
- ❑ libuv 高性能事件 IO 库。
- ❑ OpenLibm 以 libm 为基础的基本数学函数库。
- ❑ DSFMT 随机数生成库。
- ❑ OpenBLAS 线性代数 BLAS 加速库。
- ❑ LAPACK (≥ 3.5) 线性代数求解及特征值求解等加速库。
- ❑ Intel MKL (可选) 可用于替代 OpenBLAS 及 LAPACK 库。
- ❑ SuiteSparse (≥ 4.1) 基于稀疏矩阵的线性代数库。
- ❑ ARPACK 大稀疏特征值问题的求解库。
- ❑ PCRE (≥ 10.00) 与 Perl 兼容的正则表达式库。
- ❑ GMP (≥ 5.0) GNU 多精度算术库，用于支持 BigInt 类型。
- ❑ MPFR (≥ 4.0) GNU 多精度浮点库，用于支持 BigFloat 类型。

- ❑ libgit2 (≥ 0.23) 版本管理工具 Git 操作库，用于包管理。
- ❑ curl (≥ 7.50) 提供下载及代理支持，用于包管理。
- ❑ libssh2 (≥ 1.7) 支持 SSH 通道。
- ❑ mbedtls (≥ 2.2) 加密与 TLS 支持。
- ❑ utf8proc (≥ 2.1) UTF-8 编码支持。
- ❑ libosxwind 程序调用链 (call-chain) 支持。

这些众多的依赖库会在编译时自动下载，并优先编译。此过程能否顺利，与系统环境、编译环境等都有关系。此外，在编译时，还需要多种编译工具的支持，包括：

- ❑ GNU make 文件依赖构建。
- ❑ gcc & g++ (≥ 4.7) 或 Clang (≥ 3.1 , OS X 系统的 Xcode 4.3.3) C/C++ 编译与链接工具链。
- ❑ libatomic 原子操作库。
- ❑ python (≥ 2.7) 用于构建 LLVM。
- ❑ gfortran 编译链接 Fortran 库。
- ❑ perl 对库的头文件进行处理。
- ❑ wget、curl 或 fetch (FreeBSD) 用于下载外部库。
- ❑ m4 用于构建 GMP。
- ❑ awk Makefiles 处理工具。
- ❑ patch 源代码文件处理。
- ❑ cmake ($\geq 3.4.3$) 及 pkg-config 用于构建 libgit2 库。

除了要做上述这些诸多的环境准备，还需对一些选项进行恰当的配置。而且，对于 Windows 平台，还需要安装 Cygwin 环境及其中的 MinGW-w64 编译器工具链。

可见，从源码编译 Julia 程序并不会是一件轻松愉快的事情，难免会遇到各种问题。所以笔者认为，除非开发需要，例如需要进行平台移植或其他特定情况，最好不要选择编译的安装方式。因篇幅所限，关于各个平台上的编译过程，本书不做过多赘述，有兴趣的读者可参见官方资料。

1.3 交互式控制台

安装之后，在 Windows 中只需点击 Julia 程序或其快捷方式，或者在 Linux、MacOS 中的 Shell 中，运行如下命令：

```
$ julia
```

便可启动 Julia 环境，打开 Julia 提供的交互式控制台 REPL (Read Eval Print Loop)，会出现类似图 1-1 的界面。

界面中展示了官网及文档网址、帮助提示信息及版本发布等信息，随后便会出现如下提示符：

```
julia>
```

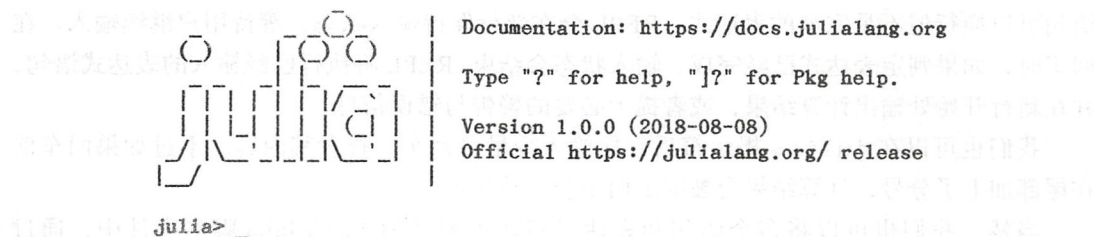



图 1-1 REPL 启动信息

这是 REPL 交互命令的输入位置，奇妙的 Julia 学习之旅便可从此开始了。

如果需要了解 REPL 启动信息之外更多的环境信息，我们便可在提示符 `julia>` 之后输入函数 `versioninfo()` 并回车：

```
julia> versioninfo()
Julia Version 1.0.0
Commit 5d4eaca0c9 (2018-08-08 20:58 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
  CPU: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-6.0.0 (ORCJIT, skylake)
```

返回的内容会因为主机的操作系统、硬件配置或 Julia 版本等不同而不同。

Windows 内置的字体对 Unicode 字符集的支持并不全面，可更换 DejaVu 字体集，能够通过 GitHub 免费获得，网址为 <https://dejavu-fonts.github.io>，可以用于替代 Windows 控制台（终端）的字体。另外，如果不希望使用 Windows 默认的控制台，也可以采用不同的终端程序，例如 Conemu 和 Mintty。当然，还可以使用交互性更好的 IDE，包括 Juno、Sublime-IJulia 或 IJulia 等。



REPL 是学习 Julia 一个非常方便的环境，我们能够直接在其中输入并执行 Julia 语句，而且可及时获得计算结果。本书中的绝大部分示例也都会基于 REPL 给出，所以建议读者在阅览本书时，最好身边有个安装 Julia 环境的电脑并已打开了 REPL 环境，能够对照着书中的例子逐一尝试，这样可以更好理解所述的内容。

当然，在学习或使用告一段落后，可在 REPL 中按下 `CTRL+D` (`^D`) 或者在提示符后输入 `quit()` 并回车，退出该环境，同时对话窗口会关闭。

为了避免在使用 REPL 中出现不必要的麻烦，下面简单地介绍一下 REPL 的使用方法，包括输入语句、快捷键、帮助等。

1. 语句输入

在 Julia 的语法中，英文分号表示语句或表达式的结束，但不是必需的。

在 REPL 中可以输入单行或者多行语句，当我们按下 `ENTER` 键回车换行时，REPL 会自动判断表达式语句是否完整。如果不是可以换行的语句，会以红色信息提示错误；如果

语句可以换行但不是完整的表达式，REPL 会在新行保持输入状态，等待用户继续输入。在回车时，如果判定表达式已经完成，输入状态会结束，REPL 将执行已经输入的表达式语句，并在新行开始处输出计算结果，或者提示必要的警告与错误信息。

我们也可以在 `julia>` 提示符后直接输入变量名回车，查看其内容。不过如果回车前在尾部加上了分号，计算结果会被屏蔽而不会自动显示。

当然，我们也可以将多个语句和表达式写在扩展名为 `.jl` 的 Julia 脚本文件中，通过 Julia 程序进行调用执行，后文会单独介绍。

2. 全局变量 `ans`

为了方便地调试各种表达式，REPL 中每段语句的执行结果都会临时存储在一个名为 `ans` 的全局变量中。即使我们将运行结果赋值给自定义变量，`ans` 中也同样会保留一份拷贝，而且其内容会在语句的每次成功运行后自动更新。

不过，该变量仅在 REPL 中有效，在 Julia 脚本文件中是没有实际用途的。

3. 内容打印

如果要在一段语句中打印中间结果，可以使用 `print()` 及 `println()` 函数。这两个函数都可以接收任意类型、任意数量的参数，在执行时会立即将参数的内容打印到屏幕上。例如：

```
julia> print(1, 2, "abc", 'a', [1 2 3], "\n", (1,2,3))    # 换行需要自己输入换行符（井
                                                         # 号为行内注释）
12abca[1 2 3]                                             # 打印的内容，第一行
(1, 2, 3)                                                  # 打印的内容，第二行
```

两个函数的区别仅在于：前者需显式地在参数中给出换行符 `\n` 才会在尾部换行打印，后者会在所有参数打印完后自动换行。



提示 在专门介绍 Julia 函数的定义与调用方式之前，我们经常会使用这两者之外的各种函数。一般而言，Julia 中的函数调用与其他语言相似，即函数名后跟着圆括号，其中以英文逗号列出所需的参数值，如上例中所示。

4. 帮助模式

在使用 REPL 的过程中，我们可以随时在 `julia>` 之后按下英文问号 `?` 所在的按键，打开帮助模式，此时提示符会发生变化：

```
help?>
```

这时便进入了帮助模式。在新的提示符之后输入需要查阅的任何内容，即可获得相关帮助信息。例如：

```
help?> Int
search: Int Int8 Int64 Int32 Int16 Int128 Integer intersect interrupt intersect!
InteractiveUtils InterruptException

Int64 <: Signed

64-bit signed integer type.
```

其中显示了有关 Int 的文档内容（此处暂不解释，通过后续学习便可了解）。



注意 按下 ? 键打开帮助模式时 julia> 之后不能有任何内容，否则会切换失败。

5. 历史查询

REPL 会记录输入过的语句表达式与当时的模式状态。只需在提示符 julia> 后按向上 (Up) 的方向键，便会出现曾经输入的语句。若不断按下该键，历史语句便会按时间从近及远地逐一呈现；而按下 Down 键时会回头反向查询。在需要的语句出现时，按下回车便可重新执行该语句。

除此之外，也可以按下 CTRL+R (^R) 复合键打开历史搜索模式。在如下的提示符之后输入字符，便能列出与之相关的历史语句：

```
(reverse-i-search)`':
```

1.4 命名规则与关键字

语言中的各种要素，包括关键字、类型、变量、函数等，都需要有标识的名字。在 Julia 中创建这些要素时，需要遵循 Julia 在命名方面的规则：

- ❑ 内置的关键字可以是名称的一部分，但不能作为完整的名称。
- ❑ 名称对大小写敏感。
- ❑ 名称首字符必须是下划线、英文 26 个字母的小写或大写，或者编码大于 0x00A0 的 Unicode 字符^①（这是 Julia 不同于其他语言的地方）。
- ❑ 名称中不能有算术运算符或内部的标识符，包括 @、#、\$、%、^、& 等。

例如，以下的名称是合适的：

```
Abc abc_cde _fg China china VAR Var!02 Var
```

其中，China 与 china 是不同的名称；虽然 Var!02 是允许的（符号 ! 是逻辑运算符），但不建议这么做。下面的名称是不符合规则的：

```
1abc @abc $var [var] for
```

另外，Julia 内置了大量的函数或常量。如果名称与它们相同，虽然语法规则上是允许的，但不会成功。例如：

```
julia> pi
π = 3.1415926535897...
```

```
julia> pi = 3
ERROR: Cannot assign variable Math Constants pi from module Main.
3
```

① Unicode 字符有详细的分类表，类别有 31 种。至于 Unicode 编码大于 0x00A0 的字符集，主要指分类为 Letter 或 Symbol 的那些字符，类别中分别有 Lu/Ll/Lt/Lm/Lo/Nl 或 Sc/So 标识，以及数学符号分类 Sm 中的部分字符。具体参见 www.fileformat.info/info/unicode/category/index.htm。

所以,从编程规范来说,也不建议选用有冲突的名称。

不过,由于 Julia 对 Unicode 的广泛支持,在命名方面我们有了更多的选择。例如:

```
中国 안녕하세요 δ
```

以上这些都可以在 Julia 中作为名称使用:

```
julia> δ = 0.00001
1.0e-5
```

```
julia> 안녕하세요 = "Hello"
"Hello"
```

```
julia> 中国 = "中华人民共和国"
"中华人民共和国"
```

这是 Julia 独具特色的地方。

无论是大肆流行的 Python 语言,还是广泛使用的 Java 语言,或是性能之王 C++ 语言,对 Unicode 的支持都极为蹩脚。对于中国这种非英文为母语的国家,一旦涉及本土语言文本的处理,就会遇到很多的麻烦。而 Julia 从根源上就考虑了多国语言问题。以此为基础,非英文处理将有着极大的便利。可以说,由于大量 Unicode 字符的加入,Julia 语言开发将极为有趣。

而且,Unicode 还只是 Julia 的冰山一角,相信读者在后续逐步学习的过程中,会被其独特的魅力所吸引。

关键字是 Julia 语言的基本元素,用于关键的声明、标识或限定,一般是一串小写字母。本书将它们粗略分成六类,列举如下:

- ❑ 类型声明: abstract、primitive、type、struct、function、macro、new。
- ❑ 权限标识: global、local、mutable、const、outer。
- ❑ 模块操作: module、baremodule、using、import、export。
- ❑ 逻辑结构: where、for、while、break、continue、if、elseif、else、in。
- ❑ 语句块: begin、quote、let、end、do。
- ❑ 混合编程: ccall。

它们的意义会在后续章节中遇见,到时会具体讲解。

1.5 先睹为快

在我们开始对 Julia 这门强大的语言进行系统的学习之前,本节给出一些简单的例子,先对 Julia 语言的表达方式有个直观的感受。

1.5.1 Hello World

一如既往地,我们给出一个语言结构最简单的 Hello World,如下所示:

```
julia> HelloWorld = "欢迎来到Julia语言的世界!"; # 定义字符串变量HelloWorld
```

```
julia> println(HelloWorld)
欢迎来到Julia语言的世界!
```

```
# 打印定义的变量
# 此行为执行后输出的内容
```

此例在 REPL 中实现，其中的 # 号及其之后的内容是笔者额外添加的注释，以便于读者理解。该例只有两句，第一句中，定义了字符串变量 HelloWorld，内容为“欢迎来到 Julia 语言的世界！”。为了清晰，该语句的尾部附加了分号，避免在回车后 REPL 自动打印该变量内容。第二句中，便是用 println() 函数将该变量的内容打印出来。

这个例子只有最基本的功能，定义变量及打印变量，用法极为简单直接。

作为动态语言的 Julia 语言，并不像 C/C++ 或 Java 语言需要 main 入口函数，所以我们在编写 Julia 程序时，只需直接提供需要运行的语句即可。

1.5.2 体型分布案例

这次给个复杂点的例子：有 1000 个人的体型样本，包括体重与身高两项指标，不考虑性别和年龄因素，计算每个人的 BMI (Body Mass Index) 指数，并根据关于肥胖的中国参考标准（见表 1-2），统计各种体型分类的人数。为了编程的方便，在表中先对 BMI 分类进行了编号，对应 1~6 类。另外，在实现时，初始的样本数据可采用随机数的方式生成。

表 1-2 体型 BMI 指数中国标准（数据来自网络）

BMI 分类编号	BMI 分类	中国参考标准	相关疾病发病的危险性
1	体重过低	$BMI < 18.5$	低（但其他疾病危险性增加）
2	正常范围	$18.5 \leq BMI < 24$	平均水平
3	肥胖前期	$24 \leq BMI < 28$	轻度增加
4	I 度肥胖	$28 \leq BMI < 30$	中度增加
5	II 度肥胖	$30 \leq BMI < 40$	严重增加
6	III 度肥胖	$BMI \geq 40.0$	非常严重增加

下面，我们采用两种不同的方式实现上述的需求：一种是基于数组的方式，另外一种基于复合结构的方式，具体见下文。由于涉及的代码会比较多，所以本例不再在 REPL 中演示，而在脚本中实现，详细源码可在 <https://gitee.com/juliaprogram/bookexamples.git> 中下载。

1. 数组实现方式

首先在磁盘中新建一个名称为 bmi_array.jl 的文件，并使用常用的文本编辑器（例如 VS Code、Notepad++、Sublime、UltraEditor、Atom 等）打开进行编辑。

第一步，使用随机函数 rand() 生成原始数据，包括 1000 个人的身高及体重样本。实现的代码如下：

```
# 使用均匀分布随机数生成1000个身高样本，取值范围为 [0,1)
heights = rand(Float64, 1000)
```

以及

```
# 使用均匀分布随机数生成1000个体重样本，取值范围为 [0,1)
weights = rand(Float64, 1000)
```

其中的 `heights` 与 `weights` 均是元素类型为 `Float64` 的数组，长度为 1000，内容类似于：

```
1000-element Array{Float64,1}:
 0.327989
 0.371755
 0.640065
 0.891165
 0.735425
 0.428819
 ... # 已省略
```

因为随机函数 `rand()` 的取值区间为 $[0, 1)$ ，所以需要转换到正常人的身高与体重范围。我们采用区间映射的方式，函数为：

$$b = \frac{(b_{\max} - b_{\min})}{(a_{\max} - a_{\min})} \times (a - a_{\min}) + b_{\min}$$

该函数可以将 $[a_{\min}, a_{\max}]$ 的任意值 a 映射到区间 $[b_{\min}, b_{\max}]$ 中的 b 值。利用此函数，可以分别将身高 `heights` 数组元素值均映射到 $[1.5, 1.8)$ 区间，将体重 `weights` 数组元素值映射到 $[30, 100)$ 区间，即身高分布在 1.5~1.8 米范围，同时体重分布在 30~100 千克，具体实现语句为：

```
# 将 身高 数据映射到 [1.5, 1.8) 米
heights = heights .* (1.8-1.5) .+ 1.5

# 将 体重 数据映射到 [30, 100) 千克
weights = weights .* (100-30) .+ 30
```

由于涉及标量与矢量（数组）的混合计算，所以采用了 Julia 特有的点操作（后面会进行深入地学习）。上述语句虽然涉及数组的逐元计算，但并没有使用循环结构，语法极为简洁直观。



提示 这两项数据呈现正态分布是较为符合实现实情况的，不过为了转换的方便，我们采用了均匀分布的 `rand()` 函数，有兴趣的读者可以更换为 `randn()` 函数，来生成初始的样本数据。

接下来，定义一个用于计算 BMI 指数的函数，如下所示：

```
bmi(w, h) = w / (h^2)
```

这是一种 Julia 式的函数定义方式，适用于实现简短的函数。其中的 `bmi` 是函数名， (w, h) 中的 `w` 和 `h` 是输入参数，分别是体重与身高值。

此后，便可基于已有的身高与体重数据计算上述 1000 个样本的 BMI 指数了，实现如下：

```
indexes = broadcast(bmi, weights, heights)
```

按常规，此处应该有循环，但是并没有。我们利用 Julia 特有的 `broadcast()` 函数，实现了将函数 `bmi()` 逐一施用于数组 `weights` 和 `heights` 元素，并能够自动取得两个数组的对应元素，自动将两对元素作为 `bmi()` 函数的输入参数，计算对应体型样本的 BMI 指数。如果一定想用一下循环结构，可以如下实现：


```

indexes = Array{Float64,1}(1000)      # 创建有1000个元素的一维数组对象
for i in 1:1000                        # 循环1000次
    indexes[i] = bmi(weights[i], heights[i]) # 成对取得体重与身高数据，计算第i个样本的BMI指数
end

```

可见，这样的方式中代码量多出了很多，远没有上述的一句话实现简洁。

更令人惊奇的是，对于 BMI 指数这种计算简单的过程，完全可以不用预先定义 `bmi()` 函数，而是采用 Julia 特有的点操作符直接实现：

```
indexes = weights ./ (heights.^2)
```

这同样能够将运算过程自动施用于 `weights` 和 `heights` 的元素中，而且仍不需要循环结构，表达极为高效、简洁。在本书之后的内容中，我们便能够深入了解这种逐元计算机制，这便是所谓的矢量化计算方法。

在 BMI 指数计算完成后，我们定义一个名为 `bmi_category` 的函数，用于对得到的指数进行分类，代码如下：

```

# 对BMI指数进行分类
# 1-体重过低，2-正常范围，3-肥胖前期，4-I度肥胖，5-II度肥胖，6-III度肥胖
function bmi_category(index::Float64)
    class = 0
    if index < 18.5
        class = 1
    elseif index < 24
        class = 2
    elseif index < 28
        class = 3
    elseif index < 30
        class = 4
    elseif index < 40
        class = 5
    else
        class = 6
    end

    class # 返回分类编号
end

```

由于该函数语句较多，所以该函数并没有采用 `bmi()` 函数定义时那种“直接赋值”的方式，而是采用了带有 `function` 关键字的常规函数定义方式，并利用 `if~elseif` 判断结构实现了对输入 BMI 指数值 `index` 进行逐层判断。

在得到最终的分类编号 `class` 之后，需要将其值返回。在 Julia 中，只需在函数结束的最后语句中直接列出该变量即可，显式的 `return` 关键字不是必需的。

然后，便可通过该函数对 `indexes` 中的 1000 个 BMI 指数进行分类了，实现语句为：

```
classes = bmi_category.(indexes) #注意函数名之后有一个小点号
```

同样采用点操作实现了数组的逐元计算。这种点操作不仅适用于上述的运算符，也同样适用于普通定义的函数。该语句执行后，会得到类似如下的结果：

```

1000-element Array{Int64,1}:
 2

```

```
5
3
1
2
1
5
...
```

```
# 其他已省略
```

最后，对 `classes` 中的类别编号进行统计：

```
# 统计每个类别的数量
for c in [1 2 3 4 5 6]
    n = count(x->(x==c), classes)
    println("category ", c, " ", n)
end
```

```
# 遍历6个类别，c为类别ID
# x->(x==c)为匿名函数
# 打印结果
```

实现中使用了 `for` 循环结构对类别编号集合进行遍历，逐一对各类型进行统计。其中的 `count()` 函数是 Julia 内置的，能够对数组中满足条件的元素进行计数，而条件由该函数的第一个参数提供。条件参数需是一个函数对象，且有一个输入参数，并需返回布尔型值。有 1 处在上述代码中，这个条件函数为 `x->(x==c)`，是一个匿名函数，等效于：

```
condition(x) = (x==c)
```

不过，因为简短，又需作为另外一个函数的参数，所以采用匿名函数的定义方式是非常合适的。

至此，需求需要实现的功能全部完成了。之后我们打开 REPL，执行以下语句：

```
julia> include("/path/to/bmi_array.jl") # 文件路径根据实际情况提供
```

便可获得最终的结果，显示的内容类似于：

```
category 1 291
category 2 221
category 3 151
category 4 77
category 5 238
category 6 22
```

```
# 体重过低共计291人
# III度肥胖共计22人
```

这里，我们总结一下：在整个实现中，数据流主要以数组结构表达，并在对数组的逐元操作中，利用 Julia 的点操作及 `broadcast()` 函数两种方式进行矢量化计算，避免了大量的循环结构，代码的实现极为简洁、高效、直观。

2. 复合类型实现方式

下面，我们再尝试另外一种实现方式。同样，在磁盘中新建一个名称为 `bmi_struct.jl` 的脚本文件，并使用文本编辑器进行编辑。

首先，定义一个复合结构类型，包括四个成员字段，分别表示某个人的身高、体重、BMI 指数及 BMI 分类编号。

```
mutable struct Person
    height # 身高，单位米
    weight # 体重，单位千克
    bmi    # 计算得到的BMI指数
    class  # 根据BMI指数计算得到的分类标识
           # 1-体重过低，2-正常范围，3-肥胖前期，4-I度肥胖，5-II度肥胖，6-III度肥胖
end
```

再定义一个集合类型，用于容纳样本数据，如下所示：

```
people = Set{Person}()
```

随后使用均匀分布生成 1000 个体型数据，并放入集合 `people` 中：

```
for i = 1:1000
    h = rand() * (1.8-1.5)+1.5 # 生成身高数据，并将其映射到[1.5, 1.8)区间
    w = rand() * (100-30)+30   # 生成体重数据，并将其映射到[30, 100)区间
    p = Person(h, w, 0, 0)     # 基于身高与体重数据创建Person对象p，
                                # BMI指数和分类编号均初始化为无效的0值

    push!(people, p)           # 将对象放入集合people中
end
```

然后定义 `bmi()` 函数，基于每个 `Person` 对象的身高及体重数据，计算其 BMI 指数并同时进行分类，代码如下：

```
function bmi(p::Person)
    p.bmi = p.weight/(p.height^2) # 计算BMI指数
    p.class = bmi_category(p.bmi) # 分类，得到类别ID，已在前文实现过
end
```

函数 `bmi()` 中原型中的 `::Person` 用于限定输入参数 `p` 变量只能是 `Person` 类型。

最后，遍历 `people` 中的 1000 个样本，对 BMI 类别分别进行统计：

```
# 对1000个样本执行BMI计算，并统计分布
categories = Dict{Int, Int}() # 字典结构，记录各类的人数
for p in people               # 遍历1000个样本
    bmi(p)                    # 计算BMI指数并分类，会直接修改p中的属性字段
    categories[p.class] = get(categories, p.class, 0) + 1 # 对p.class类的计数器累加
end
```

实现中，对 `Dict` 类型的对象 `categories` 可以两种访问方式，一种是与数组下标极为类似，用于获得类别对应的计数器，另一种是 `get()` 函数，也是用于获得类别对应的计数器内容，区别在于后者能够在 `categories` 还不存在键 `p.class` 时，也能够返回有效值（默认值 0）。

完成后，打印 BMI 分类统计的结果。方法极为简单，只需直接列出变量名：

```
categories
```

至此，功能全部实现了。打开 REPL，执行以下语句：

```
julia> include("/path/to/bmi_struct.jl") # 文件路径根据实际情况提供
```

便可获得最终结果，内容类似于：

```
Dict{Any,Any} with 6 entries:
 4 => 89
 2 => 219
 3 => 158
 5 => 234
 6 => 18   # III度肥胖共计18人
 1 => 282  # 体重过低共计282人
```

不过注意 `Dict` 中的数据是无序的，所以打印的内容中没有按照类别 ID 排列。

有别于第一种实现方式，本方式采用复合类型对数据和操作进行了封装，在业务概念或逻辑上能够显得更为条理清晰，而且整个实现过程也并不复杂。

1.5.3 小结

上述关于体型分布的例子，采用了两种方式实现，每种都各具特点。代码中已经涉及 Julia 各种主要的语法要素，包括数组、字典、集合等可迭代数集结构，以及矢量化计算、循环结构、判断结构、复合类型、函数定义的三种方式、类型限定等各种语法特性。目前是否能看懂不是最重要的，对于熟悉其他语言的读者，能够在对比中感受到 Julia 在语法表达中的与众不同，对 Julia 语言能有个直观的认知，有助于后续对其进行深入地学习。

从下一章开始，我们将逐步介绍 Julia 在数值、类型、运算符、参数化、控制逻辑、函数、多维数组及其他数集、字符串、元编程、流、并行化等各方面的内容。在后续的学习中，Julia 语言简洁、高效、直观的语法表达，高性能、高可用、高灵活性的机制，相信都不会让读者失望。希望有一天，在工业领域 Julia 能够像 Python、C++ 或 Java 那样，成为生产部署的首选，这也是笔者的愿望。

基础概念

Julia 语言是在如今计算机领域高速发展的情况下出现的，在开发之初就有了很高的起点。该语言不但借鉴了其他众多成熟、历史悠久的语言，而且融入了软件工程、编程范式等方面很多先进的设计理念。

为了尽量避免后续章节中的原理性内容给语言新学者带来困惑，也为了帮助读者更好地理解 Julia 的设计机制与语法特点，特在此章就一些基本概念进行介绍。不过对于有经验的开发者，本章的内容可以选读。

2.1 静态与动态语言

人类与计算机是两个完全不同的物种，所以在驱动计算机或机器进行工作时，需要一种相互都能懂的语言进行交流，这就是编程语言。

初始的语言是二进制码带，后来才是汇编语言，再往后出现了 C/C++ 语言等各种较高级语言。但这类语言仍需经过编译器处理为静态的二进制码，然后才能执行，而且对运行的平台有着高度的依赖性，为了保证可移植性与兼容性需要大量的基础工作。

编程语言在发展过程中，不断变得更具有人性化，让开发者能以更自然的方式与计算机交互。同时，在编译方面也有了显著的变化，出现了与静态编译不同的动态解释语言。

动态语言也称为脚本语言，是指程序编写完成后，不需预先编译，而是在运行期实时地解析并逐条执行，例如 Lua、Python、R、Matlab 等。动态语言的编写通常更为快捷方便，调试也更直观，不需要编译出所谓的可执行文件；部署时直接将脚本拷贝过去就行，在移植性方面更为方便。

除此之外，在类型声明与定义方面，也有静态类型系统或动态类型系统两种选择。静态类型系统一般要求在程序执行前编译时必须给定明确的、可计算的类型；而动态类型系

统在执行前一切都是未知的，只在执行时才确切知道类型。大多的动态语言采用的是动态类型系统，例如 Python；而静态语言一般采用静态类型系统。不过有些静态语言可通过一些技术支持类型声明时的灵活性，实现动态机制，例如 C++ 中的泛型与模板技术。

实际上，动态与静态语言在编写、调试、部署、运行等阶段都存在不少的细节差异。不过因为涉及很多底层编译器方面的知识，本书不再赘述，有兴趣的可查阅相关文献。

2.2 内存管理

编程语言是逻辑实现的载体，也是与机器交互的媒介。而逻辑的本质实际是数据流的收集、转化变换和分发转移。在物理概念上，数据在存取、计算等变化过程中，都会引发机器硬件信号的相应变化。所以编程就是在制定硬件信号的变化规则。

用语言编写出的程序在运行期间，实施这种变化规则的核心资源是处理器，而内存则是数据转化时的核心载体。所以内存管理都是任何一门编程语言的核心特性，也是机制设计中最为重要的部分。

通常而言，系统的内存划分为以下几个区域：

- ❑ **寄存器 (Registers)**。寄存器位于处理器的内部，是 CPU 可直接控制并进行存取的地方，但数量与容量都有限制，所以一般用于高频数据的缓存。该区主要由操作系统直接管理，用户的语言代码也没有直接的控制权。
- ❑ **栈 (Stack)**。栈区的存取效率仅次于寄存器，一般是连续的空间，常通过地址指针实施控制。该区的大小在程序运行前就会被设定好，但受限于操作系统及编译器的设置。在程序运行期间，编译器控制着栈区，实现资源的自动分配与释放，用于存储函数参数、局部变量、对象引用等。由于函数的调用及返回会涉及栈区内容的现场恢复等内部操作，所以大量的函数嵌套有可能会因为频繁的栈操作带来性能问题。
- ❑ **堆 (Heap)**。这是程序自己可控制的区域，也叫内存池。该区不受编译器的控制，占用多大或占用多久均可由开发者自行控制。不同的语言在堆的管理方式上会有所不同。在 C/C++ 这类语言中，对堆内存的分配与释放需要使用显式的方式，由开发者自行控制。如果不能在恰当时候释放内存，很容易导致内存泄漏；而且若释放不存在的内存区，也会引发致命的内存操作错误。不过在有垃圾回收机制的语言中，包括 Python、Julia 等，内存分配与释放是自动进行的，所以开发者不用过多关注，可将精力集中在逻辑实现上。不过超出可用内存的数据占用也同样会出现溢出错误。
- ❑ **静态 (Static) 区**。一般该区用于存储静态变量或者全局变量，即那些不需要变动位置的数据。在一些语言中，可以显式地告知编译器哪些变量是全局的，哪些变量是静态的。但该区的直接控制权在编译器而不在开发者手中。
- ❑ **常量 (Constant) 区**。程序中经常会有一些值是始终不变的，例如物理或数学系数、配置文件路径等。将这些特殊值放在一个专门的常量区，不但可以保证数据的安全性也能够提高运行效率。不过，常量区一般是运行期不可变的区域，由编译器直接控制。

除了上述的这些内存区，还有扩展存储空间，例如磁盘、固态硬盘以及U盘等。在需要时，可以使用程序对这些非RAM存储介质中的内容进行存取操作，但效率肯定不及内存，所以通常是IO（Input/Output，输入输出）操作最为耗时的部分。

除此之外，内存管理的另外一个方面是垃圾回收（Garbage Collection，GC），这是不少现代语言包括Java、Python、Julia等都支持的机制。在该机制支持下，堆中对象的分配与释放并不需要开发者显式地操作，而由系统自动执行，这样能够大大提高开发的效率及程序的安全性。一般而言，在语言内部会对创建的对象建立引用计数机制，便于GC记录跟踪内存使用的情况。当一个对象不再被有效引用时，便将其标识为垃圾，会在恰当的时候进行回收。

2.3 经典编程范式

面对需要程序化的实际业务需求时，往往要考虑很多问题。例如：

- ❑ 使用怎样的解决方案？
- ❑ 如何对问题进行分解、抽象？
- ❑ 如何对问题所属的各种概念和业务逻辑进行构架？
- ❑ 如何组织代码，又使用怎样的数据结构？
- ❑ 如何建立高效、低耦合、灵活的接口？

要解决这些问题，需要我们对业务有着深刻的理解，也考验着开发者在程序设计方面的经验与能力。不过这个阶段也是最有挑战性的阶段，充满了各种乐趣。

看待这些问题的角度和思维方式，构成了所谓的编程范式以及设计模式，也是我们在程序设计中长期积累出的哲学思考与智慧。这些在大量开发实践中总结出的经验，能够让我们在面临各种纷繁芜杂的程序设计工作时，选择最为恰当的技术方案，保证代码的灵活性、兼容性以及运行效率。

不同的范式，对最终的代码呈现和程序运行效能等方面都会产生影响。有的程序具有更高的运行效率，有的能够更灵活地适应业务逻辑的不断变化，有的在模块化、分层化、接口化方面更为方便，有的在代码复用、开发与维护成本上更高效。

在语言层面，有几种经典的范式：面向过程编程、面向对象编程、泛型编程和函数式编程等。有些语言只支持一种范式，有些则支持多种范式。不同的业务需求，可以选择适用的语言，使用合适的范式，甚至可以混用。

1. 面向过程编程

面对一个问题，最直接的方式是将解决流程罗列，并以代码逐句实现。这种线性处理过程，便是面向过程编程。该概念的明确表述始于1965年的E. W. Dijkstra，对当时来说是软件发展的一个里程碑。

面向过程的观点是任何程序都可以使用顺序、选择（if-else）、循环（for/while）



这三个基本结构进行控制，并可通过函数的形式对步骤进行局部封装与调用，所以也被称为结构化程序设计。例如，C 语言是典型的面向过程语言，虽然古老但至今仍有不可替代的用武之地。

在面向过程中，在实现顺序时，逢山开路，遇水搭桥，路一直往前铺。交代机器做事时，不会存在动作的主体，也不会存在业务主体，关注的是各种动作及其顺序；只需告诉机器先这样、再那样、然后再这样，最后进行输出。

2. 面向对象编程

面向对象设计向现实世界靠近了一步，有着与面向过程设计完全不同的设计思路，是软件工程与计算机语言发展的又一里程碑。这种编程范式思考的起点在于业务逻辑中的主体，有了更多、更高层次的抽象，体现了一种独特的世界观。

在面向对象的实施过程中，需先将各种动作或过程进行梳理、综合，抽象出各种交互主体并建立相应的一套事物模型（即对象）；而其中涉及的步骤和动作，则被剥离为这些对象之间的交互行为或者主体状态的变更操作，体现了数据的转移变换过程。

基于此，在包括 C++ 与 Java 的经典面向对象语言中，出现了常见的类、对象、封装、继承、多态等基础设施或概念。

对象是动作的实体，是逻辑的实际执行者；而类则是一种抽象概念，是对具有同样特点对象的描述。在面向对象的语言中，会将资源与动作进行封装，并赋予各种操作权限，定义出类结构。但类只是规则的集合，并不拥有实际资源，只说明同类对象具有哪些属性数据，具备怎样的行为，可以执行哪些操作，如何与外部交互。对象是类的实例化，会在运行期被分配到具体的内存区中，从而实现实际的操作或动作。

但现实世界是复杂多样的，事物之间往往存在着各种关联，也会有不同的概念层次。例如颜色与黄色、车与汽车等，前者与后者便存在着包含关系，在面向对象语言中对应的则是继承关系。通过定义抽象类，将“车”一类共有的属性或行为封装，例如启动、移动、停止等，然后再定义“汽车”、“平板车”等子类继承该抽象类，便可实现代码复用与层次控制。

但任何独立存在的事物都有其独特性，虽然都是车，但在形态、轮数、驱动方式等方面都有所差异。为此，在同属抽象类的子类中，相同的动作会需要不同的控制参数，也需要不同的具体实现。在实际执行时，虽然通过同一个动作名（函数名）执行相应的操作，但却能够有不同的表现，这便是多态。

但是在实例化的过程中，有些类不会出现多个对象，例如，某个共享的计算资源或服务，某个设备或端口，或者是通用的计算逻辑等。为了对这些单一的、仅需一份的对象进行安全的控制，可在实例化时创建类的单例，这也是设计模式中惯用的方式。

设计模式是编程经验的高度总结，在抽象层次的控制、结构关系解耦等方面给出了很多可供遵循并能落实的经典范例。

3. 泛型编程

“类”在面向对象语言中是一种类型，封装了共同的属性、数据和动作。但有时候，不



同的类型也有着相同的动作甚至执行过程，例如，人步行的行走路线和车行驶的路线，雨或雪下落时的物理过程等。

对于这种不同类的相同动作，重复实现过程确实是没有必要的。为此，泛型编程将常用的、统一的处理过程抽象出来，并能够适用于不同的类型，不但能最大限度地实现代码复用，而且逻辑的集中也为开发和维护带来了巨大的好处。不仅如此，在泛型编程中类型本身也可以作为参数，能与其他参数一起共同控制处理逻辑的实际运行。

泛型编程最初诞生于 C++ 语言，由 Alexander Stepanov 和 David Musser 提出，代表作是 C++ 的标准模板库（Standard Template Library，STL）。C++ 泛型编程之所以能在工业上取得巨大成功，得益于其高效性和通用性。

Julia 语言极大程度地借鉴了泛型编程的优点，并提供了完备的机制，其灵活的类型系统与类型参数化，都体现了泛型编程的强大魅力。

4. 函数式编程

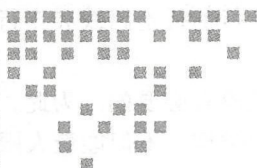
函数式编程是计算机语言在另一个层次的发展。在该范式中，函数结构不再只是执行过程的集成，而成为了类型或对象的一种。尤其在 Lisp 这样的函数式语言中，函数更是逻辑的主体和基础。

Lisp 语言是典型的函数式语言，于 1958 年由麻省理工学院的人工智能研究先驱 John McCarthy 发明。在以 Lambda 表达式及 λ 演算（Lambda Calculus）为基础的 Lisp 语言中，函数作为对象，也称为闭包（Closure）或仿函数、函子（Functor），除了可被调用外，还可以赋值、作为参数与返回值，甚至是传递转移。这种高阶函数的设计使得函数式编程具备了很多优点：简洁的代码、快速的开发、自然的表达和易于并发等。

Julia 语言同样借鉴了函数式编程的优点，不但函数是其基本的对象，类型本身、甚至代码本身都是可以操作的对象。可以说，“一切皆对象”是 Julia 最为吸引人的特色之一，使我们能够在 Julia 编程中灵活地选择想要的范式，从而适应各种不同的需求。

不过在实践中，并没有最好的范式或方案，适用的便是最好的。在合理的时间内，开发出可用、高效、稳定、可靠的程序才是唯一的检验标准与范式。





Chapter 3 第3章

数值系统

现实中的数据形式多种多样，例如数据表、文本字符、语音、图像、视频等。虽然不同的数据类型有着不同的存储方式与数据结构，但归根结底都是由一些基本的数值构成的，包括整型、浮点型、字符型等。所以数值是数据的基础类型，也是计算的基础数据。本章将详细地介绍 Julia 语言中的数值系统及特点。

3.1 整型

在数学理论及科学计算中，整数是最基本的数值类型，而整型（Integers）则是计算机中表达整数的基本语法类型。

众所周知，任何数据在计算机内部都是二进制的，而一个数值类型的表达能力与该类型的二进制位（比特，Bit）的数量有关。在 Julia 中，为了兼容不同的运行系统、节约存储空间或便于代码移植，按位数的不同对整型做了非常详细的划分，分别为 8 位、16 位、32 位、64 位甚至 128 位的整数定义了相应的整型，详见表 3-1。

表 3-1 整数类型

类 型	是否有符号	位 数	最小值（下限）	最大值（上限）
Int8	√	8	-2^7	2^7-1
UInt8	×	8	0	2^8-1
Int16	√	16	-2^{15}	$2^{15}-1$
UInt16	×	16	0	$2^{16}-1$
Int32	√	32	-2^{31}	$2^{31}-1$
UInt32	×	32	0	$2^{32}-1$





(续)

类 型	是否有符号	位 数	最小值 (下限)	最大值 (上限)
Int64	✓	64	-2^{63}	$2^{63}-1$
UInt64	×	64	0	$2^{64}-1$
Int128	✓	128	-2^{127}	$2^{127}-1$
UInt128	×	128	0	$2^{128}-1$

除此之外，为了更精确地进行整数表达，与一些强类型语言相似，Julia 中的整型又分成有符号及无符号两类，其中的有符号型能够表达负数而无符号型仅用于表达正数。而且，它们能够表述的整数范围不同，在实践中也有着不同的作用。

表 3-1 中的最后两列分别给出了各类型能够表达的整数数值范围。实际上，在 Julia 中，对于任意数值类型，我们都可以随时使用内置的函数 `typemin()` 与 `typemax()` 来获得该类型能够表达的数值范围。例如：

```
julia> typemin(Int64)      # 查看Int64类型能够表达的最小值
-9223372036854775808
```

```
julia> typemax(UInt32)     # 查看UInt32类型能够表达的最大值
0xffffffff                # 无符号整型以十六进制方式展示
```

其中的无符号整型会以十六进制的方式展示，并不影响值的实际内容。

细心的读者可能注意到，类型用作了函数的参数，这是因为类型本身也是 Julia 中可操作的对象。关于这方面的概念，后续内容会逐步地详细介绍。

3.1.1 表达方式

在开发中，经常会以字面值 (Literal) 的方式提供数据 (即直接将具体数字写在代码中)，例如：

```
julia> x1 = 20              # 20即为字面值
20
```

其中定义了 `x1` 变量，并将字面值 20 赋值给它。在赋值语句执行后，REPL 打印了执行的结果，即显示 `x1` 的结果值为 20。

但在实践中，不仅会有最为常用的十进制，还会有十六进制、八进制等。尤其是在整数表达中，多种数制的表达与转换是经常遇到的问题，甚至有时候需要直接使用二进制方式。

在 Julia 中以不同的数制输入字面值是非常方便的，只需在数值的前面设置对应的前缀即可：0x 表示十六进制数，0b 表示二进制数，而 0o 则表示八进制数 (第二个圈不是零而是字母 O 的小写)。需要注意的是，这些前缀中的字母标识 `x`、`b` 及 `o` 只能是小写，不能是大写。

下面我们分别给出数值 20 的各种进制表达方法，如下所示：

```
julia> 0x14                # 十六进制
0x14
```



```
julia> 0o24 # 八进制
0x14
```

事实上，我们可以使用 `typeof()` 函数在任何时候查看某个值或变量等任意对象的类型，例如：

```
julia> typeof(20)
Int64

julia> typeof(0x14)
UInt8

julia> typeof(0b10100)
UInt8

julia> typeof(0o24)
UInt8
```

整数类型的确定需要考虑两个方面的问题，一是有符号还是无符号，二是位数。在 Julia 自动确定字面值类型的过程中，会将十进制的字面值确定为有符号整型，而将其他进制的数均处理为无符号整型；而在位数选择方面，十进制字面值会默认与操作系统的位数保持一致，除非字面值过大，需要选择更大的位数，对于非十进制的其他数制字面值会依据值的大小选择恰当的位数。

仍以数值 20 的四种进制表达方式为例，0x14、0b10100 及 0o24 均被确定为无符号整型，而十进制被确定为有符号整型；虽然大小都是 20，但十进制选择了 64 位（笔者运行示例的系统为 64 位，后文未说明均为 64 位操作系统），其他的进制则仅选择 8 位。我们可以借助内置的 `bitstring()` 查看这些值在内部的二进制结构：

[illegible]

我们再看看一个比 20 大得多的例子：

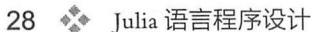


数。

做法为：

野史典故

是类型的一种



或者

```
julia> typeof(ans)
Int8
```

但如果限定的类型无法表达原始数值，Julia 会报异常，例如：

```
julia> Int64(bign)
ERROR: InexactError: trunc{Int64}(Base.OneTo{Int64}, 3000000000000000000000)
```

所以,类型的确定需要综合考虑各类型的表达能力以及内存管理方面的情况,确保在资源节约的情况下,不会导致数据的表达出现类似上述的截断性错误。

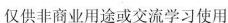
3.1.3 有无符号转换

显而易见，位数相同的整型中，与有符号相比，无符号整型能够表达更大的正数。必要时，我们可以使用 `unsigned()` 函数将有符号数值转为无符号类型，例如：

```
julia> typeof(ans)
UInt16                                     # 转为原值相同位数的有符号类型
```

反之，可以使用 `signed()` 函数将无符号数值转为有符号类型，即：

```
julia> typeof(ans)
Int8                                     # 转为原值相同位数的有符号类型
```





可以看出, 上述的 `unsigned signed()` 函数在转换时会将原值变成位数相同的无符号或有符号类型。但需要注意的是, 在这种转换中, 前者则会忽略负值的检查; 而后者会忽略表达范围越界 (Overflow, 溢出) 的情况。例如:

```
julia> a1 = Int8(-20)      # a1为8比特的整型, 负数
-20

julia> a2 = unsigned(a1)  # a2为将负数的a1转为无符号整型
0xec

julia> Int64(a2)          # a2的实际值变成了236
236

julia> bitstring(a1)
"11101100"

julia> bitstring(a2)
"11101100"
```

其中的 `a2` 在将 `a1` 转换为无符号后, 值发生了变化, 不再是 `-20` 而是 `236`。两者在存储结构中, 位序列上仍是一致的, 但类型已经不再相同, 一个是负数, 而另一个变成了正数。不过, 再次对 `a2` 进行转换时, 仍可以获得其原始的负值, 即:

```
julia> signed(a2)
-20

julia> typeof(ans)
Int8
```

再例如:

```
julia> b1 = typemax(UInt8)
0xff

julia> signed(b1)
-1      # 转换后溢出, 导致结果并非是 $2^{32}-1$ , 而变成了-1

julia> typeof(ans)
Int8
```

该例中尝试将 `UInt8` 的最大值转为有符号类型, 但显然相同位数的有符号型 `Int8` 无法表达这个值, 出现了溢出现象, 变成了 `-1` 值。

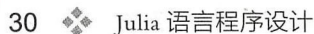


提示 为了明白其中的原委, 读者需要回顾关于计算机表达正负数的方法, 这里不赘述。需要注意的是, 这个过程 `signed()` 函数并不会上报异常, 所以在实践中需要多多留意。

在整数表达方面, Julia 提供了各种整型的表达方式, 在开发中可以根据系统特点、业务需求及内存条件, 选择恰当的类型, 以在性能与功能方面达到最优的效果。

3.2 布尔型

布尔 (Bool) 型是专门用于描述真 (true) 和假 (false) 这两种逻辑情况的特殊整



```
julia> bitstring(true)
"000000001"
```

```
julia> bitstring(false)
"000000000"
```

```
julia> bitstring(Int8(1))
"00000001"
```

```
julia> bitstring(Int8(0))
"00000000"
```

[illegible]

```
julia> Bool(UInt8(1))
true
```

```
julia> Bool(Int64(1))
true
```

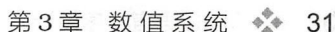
```
julia> Bool(0)
false
```

```
julia> Bool(10)
ERROR: InexactError: Bool{Bool, 10}
```

```
julia> Bool(-10)
ERROR: InexactError: Bool{Bool, -10}
```

3.3 浮点型

仅供非商业用途或交流学习使用



虽然随着现代芯片技术的迅猛发展，硬件计算能力大幅度提升，甚至有专门用于浮点运算的芯片，但浮点运算始终都是需要开发者密切关注的内容。

3.3.1 基本定义

浮点型并不像整型那样区分出有符号或无符号，而是按照表达精度分成了三类，同样与内部的位数相关，具体如表 3-2 所示。

表 3-2 浮点数类型

类 型	位 数	精 度
Float16	16	半精度
Float32	32	单精度
Float64	64	双精度

如果要在 Julia 中以字面值输入浮点数，可以有多种形式，如下所示：

```
julia> 1.0          # 正数
1.0
```

```
julia> 1.          # 省略了小数部分的0, 但保留了小数点, 用以明确是浮点数
1.0
```

```
julia> .5          # 省略整数部分的0, 保留小数点
0.5
```

```
julia> -1.23      # 负数
-1.23
```

```
julia> 1e10 ,      # 科学计数法
1.0e10
```

```
julia> 2.5e-4
0.00025
```

同样，可以使用 `bitstring()` 函数查看浮点数内部结构的二进制序列，例如：

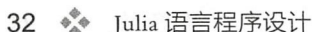
[illegible]

```
julia> bitstring(-1.23)
"1011111111110011101011100001010001111010111000010100011110101110"
```

```
julia> bitstring(2.5e-4)
"00111111100110000011000100100110111010010111100011010100111111100"
```

可见，浮点数的表达与整数有很大的不同。

默认情况下，64 位系统会自动选择 Float64 类型。若要使用其他浮点类型，则需要显式地指明，代码如下：



```
julia> Float32(0.5)      # 显式地进行类型限定
0.5f0
```

半精度类型 Float16 一般不太常用，只用于特定情况。即使将某个数值限定为 Float16 类型，参与计算时也会由 Julia 处理为 Float32 类型。

3.3.2 零的表达

浮点型在表达数值零时有些特别，存在正零和负零两种情况，它们的内存表达是不相同的，代码如下：

```
julia> 0.0 == -0.0      # 比较运算符，后文会介绍
true                   # 返回布尔型true值，表示两者的值相同
```

可见，虽然都是零，但在二进制中最高位对应的符号位却是不同的。

虽然这两种零在一般的计算中表现常常一致，但在特殊的情况，也可能会导致不同的结果。为了避免这种差异，Julia 提供了专门的函数用于生成某类型的零值，例如：

[illegible]

只要我们在需要使用零值时都采用该函数，就不会出现因为不一致而导致的无法预料的情况。

而且 `zero()` 函数除了按照指定类型生成相应的零值外，也可将某具体数值作为参数以生成其所属类型的零值，例如：

3.3.3 epsilon

实际上，由于位数所限，并不是每个实数都能在计算机中有准确的表达，计算机中的浮点型数值也只是实数集很小的子集之一。所以，计算机中相邻两个浮点型值之间存在着“缝隙”，是其无法表达的实数区域，也影响着科学计算的精度。为了能够评估这种误差，相邻浮点值间这种缝隙的宽度被称为 epsilon 值。

Julia 中可通过 `eps()` 获得某个浮点值附近的 epsilon 值，下面以 `Float64` 类型的浮点数为例：

```
julia> eps(100000.0)
1.4551915228366852e-11

julia> eps(1000.0)
1.1368683772161603e-13

julia> eps(100.0)
1.4210854715202004e-14

julia> eps(1.0)
2.220446049250313e-16

julia> eps(-1.0)
2.220446049250313e-16

julia> eps(1e-27)
1.793662034335766e-43

julia> eps(0.0)
5.0e-324
```

细心的读者能够发现，浮点值不同时 epsilon 值也不同。

实际上，随着浮点值（绝对值）从大到小，epsilon 值会呈指数级骤减，到零值处最小。而且 epsilon 值的大小关于零点对称，与浮点值的正负无关。换言之，在计算机能够表达的浮点数集合中，越靠近零点，数值的分布越稠密；而远离零点时，则会变得越来越稀疏，精度也会越来越差。

也可以直接取得某个浮点类型的 epsilon 值，代码如下：

```
julia> eps(Float16)           # 等于eps(one(Float16))
Float16(0.000977)

julia> eps(Float32)           # 等于eps(one(Float32))
1.1920929f-7

julia> eps(Float64)           # 等于eps(one(Float64))
2.220446049250313e-16

julia> eps()                  # 等于eps(one(Float64))
2.220446049250313e-16
```

不过以类型获得 epsilon 值时，上报的“缝隙”值实际是浮点值为 1.0 处的情况。若调

用该函数未提供参数, 则取默认浮点类型在值为 1.0 处的 `epsilon` 值, 例如在 64 位系统中, `esp()` 相当于 `eps(one(Float64))`。

正是由于 `epsilon` 值的存在, 计算机中的浮点数并不是无限稠密的, 而是一系列离散的数值。而且离散的浮点数是有序的, 所以可以进行逐一遍历。

内置的 `nextfloat()` 与 `prevfloat()` 函数可以分别获得某个浮点值的后继与前继浮点值。例如:

```
julia> x = 1.25f0
1.25f0

julia> nextfloat(x)
1.2500001f0

julia> prevfloat(x)
1.2499999f0
```

如果我们查看 `x` 及其前继、后继的在二进制表达, 便能够发现它们之间的大小只差一个进制位, 即:

```
julia> bitstring(prevfloat(x))
"00111111100111111111111111111111"

julia> bitstring(x)
"00111111101000000000000000000000"

julia> bitstring(nextfloat(x))
"00111111101000000000000000000001"
```

这实际也是浮点表达中 `epsilon` 值存在的重要原因。

当然, 也可以通过 `x+eps(x)` 或 `x-eps(x)` 分别获得 `x` 的后继与前继值, 代码如下:

```
julia> x + eps(x)           # 等于nextfloat(x)
1.2500001f0

julia> x - eps(x)           # 等于prevfloat(x)
1.2499999f0
```

这样的遍历操作在一些需要高精度数值迭代的场景会比较有用。

3.3.4 无穷值

在数学理论中, 常规浮点数仅仅是实数集合的一部分, 而非全部, 还有两个我们无法忽略的边界值, 即正无穷大和负无穷大。无穷值在数学上是有意义的, 在区间表达或求极限值时经常会遇到。最为常见的情况是除数为零, 此时在数学中便对应于无穷值, 但在 C++、Java 等常见的语言中会报出异常, 甚至会导致程序陡然崩溃。

但在 Julia 中, 无穷值是正常的浮点数, 有着常规浮点数同样的地位, 也是浮点型的实例对象, 同样可以参与各种计算。所以, 除零计算在 Julia 是正常的操作, 不会有任何问题。按照浮点型的不同, Julia 分别定义了三对无穷值, 具体如表 3-1 所示。

表 3-3 无穷值定义

Float16	Float32	Float64	描 述
Inf16	Inf32	Inf64 或 Inf	大于该类型所能表达的所有有限浮点值
-Inf16	-Inf32	-Inf64 或 -Inf	小于该类型所能表达的所有有限浮点值

在 64 位系统中, `Inf` 等价于 `Inf64` 而 `-Inf` 等价于 `-Inf64` 值。

需要明确的是, 不论是正无穷大还是负无穷小, Julia 中定义的无穷值都不是一种类型, 而是浮点型的数值常量。可以通过 `typemin()` 和 `typemax()` 函数查看无穷值在浮点数集合中所处的位置, 例如:

```
julia> typemin(Float64)
-Inf
```

```
julia> typemax(Float32)
Inf32
```

```
julia> typemin(Float16)
-Inf16
```

可见它们与前文定义的常规浮点型同处于实数轴上, 而且是对应着浮点类型表达范围的极限。

有了对无穷值的支持, Julia 中的浮点运算就可以正常地使用除零操作了, 例如:

```
julia> 1.2/0
Inf
```

```
julia> -1.2/0.0
-Inf
```

```
julia> Float32(9)/zero(Float16)
Inf32
```

```
julia> Float16(9)/zero(Float64)
Inf
```

```
julia> typeof(ans)
Float64
```

实际上, 这种操作在数学中也是有意义的。

如果要获知某个变量或值是否是无穷值, 可通过 `isfinite()` 或 `isinf()` 函数进行判断, 例如:

```
julia> isfinite(Inf)
false
```

```
julia> isinf(-Inf)
true
```

```
julia> isfinite(3.0)
true
```

```
julia> isinf(2.5e-10)
false
```

关于无穷值在计算中的更多应用，我们会在后续的章节中介绍。

3.3.5 非数值

基于浮点数的计算中，往往会出现无效的浮点值。这样的事物不是任何浮点数，但我们需要标记它的存在，这种情况在大规模数据操作中经常会遇到。为了应对这种情况，Julia 另外定义了一组特殊的浮点值常量，即“非数值 (Not-a-Number)”。

定义的三种浮点类型（即 Float16、Float32 及 Float64）对应的“非数值”分别是 NaN16、NaN32 及 NaN64（在 64 位系统中等价于 NaN）。

可用 isnan() 来检验一个变量或数值是否为“非数值”，例如：

```
julia> isnan(10)
false
```

```
julia> isnan(-NaN)
true
```

```
julia> isnan(NaN16)
true
```

为了叙述方便，后文涉及“非数值”这类特殊浮点值时，均以 NaN 指代。

3.3.6 内置常量

为了方便，在 Julia 的 Base.MathConstants 模块中定义了常见的数学或物理等方面的系数常量，如表 3-4 所示。开发者可以不用自己重复定义，直接使用即可。

表 3-4 常用的内置常量

系数常量	对应的变量名称	取值
圆周率	pi 或 π	3.1415926535897...
自然常数 (欧拉数)	ε 或 e	2.7182818284590...
卡特兰数	catalan	0.9159655941772...
欧拉常数	eulergamma 或 γ	0.5772156649015...
黄金比例	φ 或 golden	1.6180339887498...

注意“变量名称”中的希腊字母并不是某种数学表达，而是 Julia 中的变量名。其中 π 、 ε (不是英文字母 E 的小写)、 γ 和 φ 都是希腊字母，但都是正常 UTF-8 字符构成的 Julia 变量，可以像其他变量一样使用，例如：

```
julia> using Base.MathConstants # 引入常量模块，后面会介绍
```

```
julia>  $\pi$ 
 $\pi$  = 3.1415926535897...
```

```
julia> φ  
φ = 1.6180339887498...
```

```
julia> γ  
γ = 0.5772156649015...
```

```
julia> 2π  
6.283185307179586
```

```
julia> γ+1  
1.5772156649015328
```



提示 至于这些字符的输入方式，Julia 提供了反斜线转义的方式，并在官方文档中提供了详细的列表。不过转义方式仅适用于 REPL 环境，不适用于脚本环境。如果在脚本中需要，直接复制粘贴或许更为方便。

不过深究的读者会发现，这些系数常量并不是浮点类型，即：

```
julia> using Base.MathConstants
```

```
julia> typeof(π)  
Irrational{π}
```

```
julia> typeof(φ)  
Irrational{φ}
```

```
julia> typeof(γ)  
Irrational{γ}
```

其中的 `Irrational` 便是 Julia 内部定义的无理数类型。

无理数是不能写作整数比值的无限不循环小数，也是实数的一种。因为我们一般不需要主动创建无理数，而且 `Irrational` 型可以像浮点型那样操作，所以本书不对该类型赘述。

3.4 有理数型

有理数与无理数相对，也是实数集合的子集，可以简单地认为是分子与分母均为整数的分数。

在 Julia 中直接提供了这种数值的表达方式，其原型（其中涉及的抽象类型、类型参数化等概念会在后续章节中详述）为：

```
Rational{T <: Integer} <: Real
```

其中，`Rational` 是有理数的类型名；`Integer` 是 Julia 中定义的抽象类型（后文介绍），用于限定分子与分母都必须是上文中 `Int8`、`Int16` 等具体整数类型的一种；`Real` 是抽象类型，表达的是 `Rational` 同浮点型、无理数型一样，都是实数的一种。

`Rational` 类型是 Julia 中特有的，以“分子 // 分母”格式表达。其中双斜线并非是运算符，而是表达有理数结构的特定操作符。例如：


```
julia> 2//3          # 分子与分母都是Int64类型
2//3

julia> typeof(ans)
Rational{Int64}

julia> Int8(2)//Int32(7)  # 分子是Int8类型，而分母是Int32类型
2//7

julia> typeof(ans)
Rational{Int32}          # 类型被统一为Int32
```

可见，分子与分母为不同整型时，Julia 会通过必要的隐式转换，将两者的类型进行统一。而且创建的 Rational 数值在 Julia 内部会被约分为标准形式，确保分母不为负数，例如：

```
julia> UInt32(2)//Int8(10)
1//5

julia> 5//25          # 被约分
1//5

julia> 1//-2          # 负号被放于分子上
-1//2

julia> -2//4
-1//2
```

如此一来，数学意义上相等的有理数，在语言内部的表达便是唯一的。

由于对无穷值的支持，所以在 Rational 对象中允许分母为 0，但不允许分子和分母同时为 0，例如：

```
julia> 5//0
1//0

julia> -3//0
-1//0

julia> typeof(ans)
Rational{Int64}

julia> 0//0
ERROR: ArgumentError: invalid rational: zero(Int64)//zero(Int64)
```

其中，前两个除零有理数定义成功，且都被约分为同一形式；但后一个因为分子和分母都为 0，所以创建失败。

显然，如果分子或分母中任意一个为浮点值，创建 Rational 数值对象也会失败，例如：

```
julia> 1.2//3
ERROR: MethodError: no method matching //(::Float64, ::Int64)

julia> 2//3.1
ERROR: MethodError: no method matching //(::Int64, ::Float64)
```

因为 Rational 的声明中限定了分子、分母的类型必须是整型的一种。

当然, `Rational` 类型的有理数值均可转换到浮点值, 例如:

```
julia> Float64(3//2)
1.5

julia> Float64(-4//-12)
0.3333333333333333

julia> Float64(5//0)
Inf # 分母为0的正有理数, 对应正无穷

julia> Float64(-3//0)
-Inf # 分母为0的负有理数, 对应负无穷
```

同时, 浮点值也能够转换到 `Rational` 数值, 例如:

```
julia> Rational(1.5)
3//2

julia> Rational(0.3333333333333333)
6004799503160601//18014398509481984

julia> Rational(-2.5)
-5//2

julia> Rational(Inf)
1//0

julia> Rational(-Inf)
-1//0
```

其中, 为何 `0.3333333333333333` 转换的结果不是 `1//3` 而是由一大串数字构成的 `Rational` 类型对象, 有兴趣的读者可以研究下。

如果要获得 `Rational` 对象的分子、分母部分, 可以分别通过 `numerator()` 和 `denominator()` 函数实现, 例如:

```
julia> numerator(2//3)
2
julia> denominator(2//3)
3
```

3.5 复数型

复数与有理数类型一样, 也是一个二元数, 在高等数学与信号处理中经常涉及。Julia 也内置了复数型, 并提供了各种常用的运算。

复数同样是一个参数化的类型, 其原型为:

```
Complex{T <: Real} <: Number
```

其中, `Number` 是抽象类型, 表示 `Complex` 是数值类型的一种; 而 `{T<:Real}` 表示实部与虚部的类型 `T` 可以是 `Real` 类型的任一种, 包括各种浮点型、整数型、有理数 `Rational` 型及无理数 `AbstractIrrational` 型 (`Irrational` 的父类型) 等。

特别地，当 T 取 `Float16`、`Float32` 及 `Float64` 时，对应的复数型分别为 `Complex{Float16}`、`Complex{Float32}` 及 `Complex{Float64}`。为了方便，Julia 为它们提供了别名，即 `ComplexF16`、`ComplexF32` 和 `ComplexF64`。

在创建 `Complex` 数值对象时，也可以像整型、浮点型、有理数型那样以字面值的方式提供，格式为“实部 + 虚部 `im`”。为了避免歧义，其中的虚部标识采用了 `im`，对应于数学中的虚部单位 i ，例如：

```
julia> 1 + 2im
1 + 2im
```

如上所示，虚部可由数值紧跟 `im` 构成。

若没有提供虚部值，会默认为 1，例如：

```
julia> 1+im
1 + 1im
```

若不提供实部，则会默认为 0，例如：

```
julia> 2im
0 + 2im
```

但如果使用变量提供虚部值，则需将变量与虚部单位明确的采用乘号表示出来，例如：

```
julia> a = 1; b = 2;
```

```
julia> a + b*im
1 + 2im
```

```
julia> 1 + Inf*im
1.0 + Inf*im
```

```
julia> 1 + NaN*im
1.0 + NaN*im
```

但是这种用乘号表示的复数并不推荐，因为很容易与常规的乘法冲突，所以建议使用函数 `complex()` 构造复数，例如：

```
julia> complex(a, b)
1 + 2im
```

该函数创建的 `Complex` 类型会由 Julia 自动确定。

当然也可以使用复数类型声明的构造方法（后面会介绍）创建 `Complex` 类型，例如：

```
julia> ComplexF32(1.0, 2.0)
1.0f0 + 2.0f0im
```

这种方式可以限定创建对象的类型。

正如类型的声明所述，除了整型与浮点型外，其他类型的实数也是可用于创建 `Complex` 对象的，例如：

```
julia> using Base.MathConstants
```

```
julia> complex(γ, π)
```




```
0.5772156649015329 + 3.141592653589793im

julia> typeof(ans)
Complex{Float64}           # 无理数Irrational型变成了浮点型

julia> complex(1//2, 2//3)
1//2 + 2//3*im

julia> typeof(ans)
Complex{Rational{Int64}}
```

在数学上，复数有着各种操作，比如求共轭，取绝对值，算模的值或相位角等，这些操作在 Julia 中也都可以实现，如表 3-5 所示。

表 3-5 复数的操作

函 数 名	功 能
real()	取实部
imag()	取虚部
reim()	同时取虚部、实部
conj()	复数共轭
abs()	模
abs2()	平方模（二阶范数）
angle()	相位角（弧度）

3.6 随机数

随机地生成数值，在许多场景中会用到，比如模拟中奖这种随机事件，或者蒙特卡罗 (Monte Carlo) 仿真等。但通过计算机生成随机数是一件很有挑战性的事，因为计算机的特点是一切都需要是明确的。科学家们发明了多种方法，制作出了各种伪随机数发生器 (Random Number Generator, RNG)，其中的马特赛特旋转演算法[⊖]是 Julia 默认采用的方法。

 **提示** 如果开发者不愿意使用该 RNG，可以借助 AbstractRNG 类加入新的 RNG，以便使用。

另外，因为随机数模拟的实际是某个随机事件，所以也与统计分布相关，即大量的随机数序列会呈现某种数学分布特性，比如均匀分布、正态分布或指数分布等。Julia 能够按照要求生成特定分布特性的随机数序列，并且还可以支持不同的数值类型，下面以浮点型为例，分别介绍三种常见分布的随机数生成方法。

⊖ 马特赛特旋转演算法 (Mersenne Twister Library)，由 Makoto Matsumoto (松本) 和 Takuji Nishimura (西村) 于 1997 年开发。该方法基于有限二进制字段上的矩阵线性再生，快速产生高质量的伪随机数。



1. 均匀分布随机数

函数 `rand()` 用于生成均匀分布的随机数。对于浮点类型来说，该函数生成的随机数所处的区间是 $[0, 1)$ ，即小于 1 大于等于 0 的浮点数。例如：

```
julia> rand()
0.003594517966796884

julia> rand()
0.05602819264146208

julia> rand()
0.8497003282422178
```

如上所示，按照相同的方式反复地调用同一个函数，每次都会获得不同的数值，即获得了需要的随机序列。

2. 正态分布随机数

函数 `randn()` 则是用于生成均值为 0，方差为 1 的正态分布中的随机数，例如：

```
julia> randn()
1.2632927170077926

julia> randn()
-1.2236714132749305

julia> randn()
0.4294006067651739
```

3. 指数分布随机数

函数 `randexp()` 用于生成符合指数分布的随机数，例如：

```
julia> randexp()
1.6622581377218693

julia> randexp()
0.768276340842417

julia> randexp()
0.3918476058681037
```

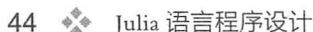
关于随机生成函数的更多用法，可参见相关文档，这里不再赘述。

3.7 任意精度算术

不论是浮点型还是整型，遇到“超大”的数值时，都会面临越界溢出的问题。例如：

```
julia> x = typemax{Int64}
9223372036854775807          # Int64能表达的最大正数值

julia> x + 1
-9223372036854775808
```



其中，x 已是 Int64 能够表达的最大数值，一旦对其 +1 便出现“反转”的情况，变成了 Int64 的最小数值。若不能跟踪这种异常，未采取恰当的处置，便会引发不可预料的重大的错误。

当使用 `BigInt` 类型时，处理上例中变量 `x` 的情况，不会再出现溢出问题，例如：

可见，取得了累加后的正确值。当然，对于更大的整数值同样如此，例如：

其中的 `parse()` 函数用于将字符串转为数值类型，在字符串学习时会详细介绍。

对于浮点数，BigFloat 同样可以解决表达范围越界的问题：

或者，可采用 `big()` 函数将不同类型的“超大”数值转为 `BigInt` 或 `BigFloat` 类型，例如：

采用 BigFloat 后，在 epsilon 值方面也有着更好的提升，例如：

可见, BigFloat 的 epsilon 值显然要比 Float64 类型的小得多。

另外，在代码中以字面值的方式输入非常大（通常也会比较长）的数值时，可以使用 Julia 特有的下划线分割方式，以便于视觉上的划分。例如：

⊖ 多精度算术库 (GNU Multiple Precision Arithmetic Library, GMP), 参见 <https://gmplib.org>.

⊖ 多精度浮点计算库 (Multiple-Precision Floating-point computations with correct Rounding, MPFR), 参见 <http://www.mpr.org>。



```
julia> 988_765.000_1234_05
988765.000123405
```

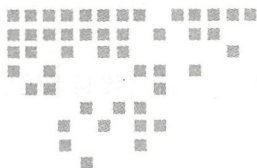
其中的下划线位置并没有明确的限制，而且同样适用于整型，且与数制无关。例如：

```
julia> 10_000_000
10000000
```

```
julia> 0xdeaf_beef
0xdeafbeef
```

```
julia> 0b1011_0010
0xb2
```

可见非常直观方便。



运 算 符

在基本的数学理论中，数值集合通常会与运算规则一起定义，形成所谓的空间或者数域，例如实数域 R 、有理数域 Q 等。其中的运算规则，是指加、减、乘、除这些基本的四则运算，也称为算术运算符，是百分比、平方根、幂乘、对数等更复杂运算的基础。

在计算机语言中，除了基本的算术运算符外，还会提供对更多运算符的支持，包括位运算符、逻辑运算符、比较运算符、更新运算符等。

本章将介绍这些运算符，为了叙述的方便，我们首先定义几个变量，用于后文中示例的常用数值：

```
a = Int64(50)      # 值为50的Int64整型
b = Int32(2)        # 值为2的Int32整型
c = Float32(2.5)    # 值为2.5的Float32浮点型
d = Float64(3.0)    # 值为3.0的Float64浮点型
x = 3+2im           # 复数型 Complex{Int64}
y = 15//8           # 有理数型 Rational{Int64}
```

4.1 算术运算符

对于基本的数值类型，包括各种整型、浮点型、有理数型、复数型等，表 4-1 中的运算符除了一些特殊计算，其他均是支持的。

表 4-1 算术运算符

使用示例	名 称	描 述
<code>+x</code>	一元加号	不变操作
<code>-x</code>	一元减号	反转 x 的符号



(续)

使用示例	名 称	描 述
$x+y$	二元加号	加法
$x-y$	二元减号	减法
$x*y$	乘号	乘法
x/y	除号	除法
$x \div y$	欧式除号	获得整数商
$x \setminus y$	被除号	等价于 y/x
x^y	幂	x 的 y 次方
$x \% y$	余	求 x 在 y 下的余数
\sqrt{x}	平方根	负数时需为复数形式，等效于 <code>sqrt()</code>
$\sqrt[3]{x}$	立方根	等效于 <code>cbrt()</code> 函数

表中列举的都是一些常规的运算符，除了被除号。提供被除号运算符的目的是为了能够更广泛地支持数学中的表达方式，使得在表达数学公式时能够更为方便，这也是 Julia 独具匠心的设计。另外，除法除了使用斜杠 (/) 表达外，还可以使用 \div 这种传统的数学符号，这是非常有意思的。

在这些运算符中，只有两个是一元运算符，即运算操作只涉及一个值或变量。例如：

```
julia> +d
3.0


julia> +y
15//8
```

一元加号不会对原值做出改变，而一元减号则会反转原值的正负表达：

```
julia> -a
-50

julia> - -a          # 两次换符号（中间有空格）
50

julia> -x             # 复数的实部与虚部值，都被反转符号
-3 - 2im
```

 注意 还有，对于复数来说，如果使用一元减号，实部与虚部会被同时改变。

再看基本的二元加法操作，代码如下：

```
julia> a + b          # Int64 + Int32
52

julia> typeof(ans)
Int64
```




```
julia> b + c          # Int32 + Float32
4.5f0

julia> typeof(ans)
Float32

julia> a + c          # Int64 + Float32
52.5f0

julia> typeof(ans)
Float32
```

其中涉及的两个值均有着不同的类型，而结果一般是两者中“较大”的那个类型。所谓较大，指该类型的表达能力比另外一个强。如上例中，整型没有小数点，所以无法准确地表达浮点数；但反之，浮点型可以精确表达整数，所以，当整型与浮点型相加时结果会是浮点类型。

下面是操作数中有复数或有理数的一些例子：

```
julia> a - y          # Int64 - Rational
385//8              # Rational

julia> d + y          # Float64 + Rational
4.875              # Float64

julia> d - x          # Float64 - Complex
0.0 - 2.0im        # Complex

julia> x + y          # Complex + Rational
39//8 + 2//1*im     # 复数的实部与虚部的类型均是有理数型
```

可见，这些基本运算符对复数型及有理数型有着自然的支持。而且，结果类型的选择同样遵循“较大原则”，即选择表现能力强的类型。不难理解，复数型比有理数型的表现力更强，而有理数型是无法表达复数的。

对于较大原则，一般而言，有理数型优于整型，浮点型优于有理数型，复数型优于浮点型。但当复数与有理数进行运算时，结果却是 `Complex{Rational{Int64}}` 型。看起来复杂，实际就是复数型，只不过其实部与虚部的类型是 `Rational{Int64}` 这个有理数型而已。

我们再看以下乘除法的例子：

```
julia> a * b          # Int64 * Int32
100                # Int64

julia> a * c          # Int64 * Float32
125.0f0            # Float32

julia> a * y          # Int64 * Rational
375//4             # Rational

julia> a / x          # Int64 / Complex{Int64}
11.538461538461538 - 7.6923076923076925im

julia> typeof(ans)
Complex{Float64}
```



结果类型同样遵循着“较大原则”。其中最后的例子稍显特别，虽然操作数的类型分别是 `Int64` 和 `Complex{Int64}`，都是以整型为基础的，但结果类型却是 `Complex{Float64}`。实际上，这并没有突破“较大原则”，但类型参数却发生了变化，不再以整型为基础，这是因为运算符是除法，所以会自动采用浮点类型。比如：

```
julia> a/b          # Int64 / Int32
25.0
```

```
julia> typeof(ans)
Float64
```

这个例子中两个操作数均是整型，且能够整除，但结果并没有保持整型，同样采用了浮点类型。这点与 C++ 等语言是不同的。

不过，在除法运算符中，斜杠的除号 `/` 与数学除号 `÷` 在对整数进行计算时，有着显著的差异，这点需要特别注意。例如：

```
julia> x = Int64(12);
```

```
julia> x ÷ 2
6
```

```
julia> typeof(ans)
Int64          # 仍为整型
```

```
julia> x / 2
6.5
```

```
julia> typeof(ans)
Float64        # 自动为浮点型
```

另外，因为特殊浮点值——无穷值的存在，所以在 Julia 中如果出现除数为零的情况，是可以顺利执行的，例如：

```
julia> c/0          # Float32 / 0
Inf32
```

```
julia> y/0          # Rational / 0
1//0
```

```
julia> x/0          # Complex / 0
Inf + Inf*im
```

但对于求余数的操作，除数是不允许为零的，例如：

```
julia> a % 0
ERROR: DivideError: integer division error
```

不难理解，这是因为除数为 0 时的求余数运算，在数学上也是无法解释的。

不过，求余操作并不仅限于整型，例如：

```
julia> c % b        # Float32 % Int32
0.5f0
```

```
julia> d % c        # Float64 % Float32
```



0.5

```
julia> y % d          # Rational{Int64} % Float64
1.875
```

但是，求余不适用于复数。

在算术运算符中，还有一种幂运算，同样适用于多种数值类型，例如：

```
julia> 2^5           # Int64
32

julia> 2.1^5         # Float64
40.841010000000001

julia> (2//3)^5      # Rational{Int64}, 注意圆括号不可少（优先级的缘故）
32//243             # 仍为原类型

julia> (2+3im)^5     # Complex{Int64}
122 - 597im

julia> (2+3im)^4     # Complex{Int64}
-119 - 120im

julia> 3im^5         # Complex{Int64}
0 + 3im             # 值未发生变化
```

当然复数的幂有些特别，但仍遵循着数学定义。

令人惊讶的是，幂运算中还支持指数为小数或复数等其他类型，例如：

```
julia> 2.1^1.5       # Float64
3.043189116699782

julia> (2//3)^1.5    # Rational{Int64}
0.5443310539518174

julia> (2+3im)^1.5   # Complex{Int64}
0.6603660276815664 + 6.814402636366297im

julia> 2.1^(2+3im)   # 指数为复数
-2.6864471968673858 + 3.497299166277224im

julia> (2+3im)^(3//2) # 指数为有理数型
0.6603660276815664 + 6.814402636366297im

julia> (2+3im)^(2+3im) # Complex{Int64}
0.6075666647314786 - 0.3087560180979021im
```

另外，指数也可以是负数，例如：

```
julia> 4.0^-2        # Float64
0.0625

julia> (2//3)^-2     # Rational{Int64}
9//4

julia> (2.0+3.0im)^-2 # Complex{Float64}
-0.029585798816568053 - 0.07100591715976332im
```


与求幂相对的，运算符 $\sqrt{\quad}$ 和 $\sqrt[3]{\quad}$ 可分别用于开平方和开立方（求立方根）。其中的 $\sqrt[3]{\quad}$ 适用于负数，但 $\sqrt{\quad}$ 不适用于负实数，如果要进行操作，负数需以复数的形式提供。例如：

```
julia>  $\sqrt{-2.0+0im}$ 
0.0 + 1.4142135623730951im
```

因为负数的根是复数，所以也要求对负数求根时必须输入复数（虚部为 0 值）。

4.2 位运算符

无论怎样的数值类型，在计算机中都是连续的二进制串。任何的运算实质都是在改变或转移这些二进制串。对数值进行位运算，包括与（And）、或（Or）、非（取反，Not）、异或（Xor）、左移及右移等时，实际是对数据的内存结构进行直接的操作。在一些情况下，直接控制这些二进制串，可以在节约计算资源或提升关键代码性能等方面带来好处。这样的操作常见于通信中的传输协议、编解码或密码学等领域。

需要说明的是，Julia 中提供的位运算符仅适用于整型，不能是其他类型，包括浮点型或有理数型等。表 4-2 给出了 Julia 位运算符的示例用法及简单的说明。

表 4-2 位运算符

使用示例	名称	描述
$\sim x$	非（取反）	1 变 0, 0 变 1
$x \& y$	与	对应位全为 1 才取 1 否则取 0
$x y$	或	对应位全为 0 才取 0 否则取 1
$x \times y$	异或	对应位不同取 1 否则取 0
$x \gg> n$	逻辑右移	不考虑符号位。右移 n 位，左侧补零
$x \gg n$	算术右移	考虑符号位。先右移 n 位，若符号位为 1，左侧补 1，否则补 0
$x \ll n$	逻辑或算术左移	左移 n 位，右侧补 0

其中除了取反是一元的，其他都是二元运算符。

1. 与或非运算

我们以实例首先了解取反操作，例如：

```
julia> m = Int8(50)
50

julia> bitstring(m)           # m值的内部二进制结构
"00110010"

julia> n =  $\sim m$ 
-51

julia> bitstring(n)           # 取反后的二进制结构
"11001101"
```

变量 `m` 的内容是一个值为 50 的有符号整数。经过取反后，其内存中的每一个二进制位均被反转，甚至包括符号位。所以取反的结果是数值 50 变成了 -50 这个负值。

再看一个无符号数值的例子：

```
julia> m = UInt8(3)
0x03

julia> bitstring(m)
"00000011"

julia> n = ~m                # 对无符号数50取反
0xfc

julia> bitstring(n)
"11111100"
```

其中值为 3 的 8 位无符号整型 `m` 的各二进制位同样被逐位反转。因为是无符号的，所以不存在正负值变换的说法，但我们可以通过将其转换到 `Int64` 类型看一下值的变化：

```
julia> Int64(m)
3

julia> Int64(n)
252
```

这是显而易见的。至于为何是“显而易见”，读者可以研究一下。

对于与、或及异或运算，在符号变化方面有些不同，例如：

```
julia> m = Int8(10);
julia> n = Int8(-7);

julia> bitstring(m)                # m的二进制结构
"00001010"

julia> bitstring(n)                # n的二进制结构
"11111001"

julia> p = m & n                    # 与操作
8

julia> q = m | n                    # 或操作
-5

julia> r = m ~ n                    # 异或操作
-13

julia> bitstring(p)                # 与结果的二进制结构
"00001000"

julia> bitstring(q)                # 或结果的二进制结构
"11111011"

julia> bitstring(r)                # 异或结果的二进制结构
"11110011"
```

这三个运算同样会涉及符号位，但结果的符号是可以根据运算的特点进行推断的。例如，正数与负数做与运算时，结果肯定是正数（因为符号位必然是 0）；而做或运算时，则会得到负数，因为符号位必然是 1。

2. 移位运算

同样，左移运算也会涉及符号位，例如：

```
julia> bitstring(m)
"00001010"

julia> bitstring(m << 1)          # 左移1位
"00010100"

julia> bitstring(m << 2)          # 左移2位
"00101000"

julia> bitstring(m << 3)          # 左移3位
"01010000"

julia> m << 3                      # 左移3位后的取值
80
julia> bitstring(m << 4)          # 左移4位
"10100000"

julia> m << 4                      # 左移4位后的取值
-96
```

随着值为 10 的 m 变量的二进制位不断左移，值会越来越大，但当移动了 4 个位置后，符号位由 0 变成了 1， m 的值也变成了负数。

但右移运算中，Julia 区分了是否考虑符号位的情况，分为算术右移及逻辑右移。在算术右移后，数值的正负是不变的，例如：

```
julia> n                          # n为负数
-7

julia> bitstring(n)
"11111001"

julia> n >> 1                      # 右移1位
-4

julia> bitstring(ans)
"11111100"
```

变量 n 是负数，符号位是 1，算术右移 1 位后，右侧的二进制位消失，但因为考虑了符号位，所以左侧在补充二进制位时，补上的并不是 0 而是 1，从而保持了符号的表达。

继续对 n 进行算术右移，会发现：

```
julia> bitstring(n)               # n的二进制结构
"11111001"

julia> bitstring(n >> 1)          # 算术右移1位
```



```

"11111100"

julia> bitstring(n >> 2)      # 算术右移2位
"11111110"

julia> bitstring(n >> 3)      # 算术右移3位
"11111111"

julia> bitstring(n >> 4)      # 算术右移4位
"11111111"

julia> n >> 4                  # 算术右移4位后的取值
-1

```

当移动了一定的数位后，二进制串中不再有 0，结果的值便保持了 -1，再算术右移任意的位数，该值都不再变化。

而对于正数来说，不断右移的结果为：

```

julia> m
10

julia> bitstring(m)           # m的二进制结构
"00001010"

julia> bitstring(m >> 1)      # 算术右移1位
"00000101"

julia> bitstring(m >> 2)      # 算术右移2位
"00000010"

julia> bitstring(m >> 3)      # 算术右移3位
"00000001"

julia> bitstring(m >> 4)      # 算术右移4位
"00000000"

julia> bitstring(m >> 5)      # 算术右移5位
"00000000"

julia> m >> 5                  # 算术右移5位后的取值
0

```

右移了一定位数后，结果值保持为 0 值，而且不再变化。

正数的这种情况在逻辑右移中也是一样的，例如：

```

julia> bitstring(m >>> 1)     # 逻辑右移1位
"00000101"

julia> bitstring(m >>> 2)     # 逻辑右移2位
"00000010"

julia> bitstring(m >>> 3)     # 逻辑右移3位
"00000001"

julia> bitstring(m >>> 4)     # 逻辑右移4位

```



```
"00000000"
```

```
julia> bitstring(m >>> 5)      # 逻辑右移5位
"00000000"
```

对负数来说，逻辑右移的效果类似，但是有所不同，例如：

```
julia> n
-7
```

```
julia> bitstring(n)
"11111001"
```

```
julia> p = n >>> 1            # 逻辑右移1位
124
```

```
julia> bitstring(p)
"01111100"
```

对于值为 -7 的 n 而言，仅移动了一位，结果的值便变成了 124 这个正数。

一般而言，任何负数只要经过一次逻辑右移便会变成正数，这是因为计算机表达正数的符号位是左侧的 0，而一旦逻辑右移，1 便会被 0 补上。



提示 在位运算方面，很多应用还会涉及循环左移或循环右移操作，有兴趣的读者可以查阅相关资料进行了解。

4.3 更新运算符

所谓更新运算（Update），实际是在算术或位运算的基础上，编程语言提供的就地（In-place）改变原有值的一种操作方法。这种更新运算符一般是由赋值操作符 = 结合算术或位运算符构成，可同时完成修改与赋值的操作，不过更新操作均是以二元运算符为基础，不支持一元运算符。

Julia 中支持的更新运算符包括：

```
+= -= *= /= \= ÷= %= ^= &= |= = >>= >>= <<=
```

它们的意义其实很明了，赋值符号 = 之前的运算符便是更新的具体操作。下面给出一些例子：

```
julia> a
50
```

```
julia> a += 1                  # a加1后，替换a的原有值
51
```

```
julia> a
51
```

```
julia> a >>= 2                # a逻辑右移2位后，替换a的原有值
```



12

```
julia> a
12
```

关于更新运算符的使用，不多作介绍，读者可以通过 REPL 多做些尝试，以进行更深入的学习。

4.4 比较运算符

数值类型对象之间的对比，也是常常遇到的问题。比较运算符便是对两个操作数的关系进行比较，在大于小于、是否相等或者是否完全相等等条件下给出逻辑判断，并返回一个布尔型的结果（Bool 值）：在给定条件为真时返回 true，为假时则返回 false 值。因为涉及两个值，所以比较运算符都是二元的。Julia 提供的比较运算符如表 4-3 所示。

表 4-3 比较运算符

运 算 符	名 称
==	等于
!= 或 ≠	不等于
≈	约等于
≉	不约等于
===	完全相同
!== 或 ≠	不全相同
<	小于
<= 或 ≤	小于等于
>	大于
>= 或 ≥	大于等于

其中有些是非传统的 Unicode 符号，例如 ≠、≤ 及 ≥ 等，这些是在数学表述中常见的符号，但在编程语言中却是不常见的，这是 Julia 的特色。

1. 大小比较

我们先看小于 (<) 及大于 (>) 这两个比较运算符，仍是从例子开始：

```
julia> a < b      # Int64(50) < Int32(2)
false

julia> a > b      # Int64(50) > Int32(2)
true

julia> c < d      # Float32(2.5) < Float64(3.0)
true

julia> a > c      # Int64(50) > Float32(2.5)
```




```

false

julia> a < y                                # Int64(50) < Rational{Int64}(15//8)
false

julia> c > y                                # Float32(2.5) > Rational{Int64}(15//8)
true

```

可见，比较运算符也可以对不同的操作数类型进行操作。

不过需要注意的是，大小的比较对两个复数是不适用的，例如：

```

julia> 3+2im < 3+1im
ERROR: MethodError: no method matching isless(::Complex{Int64}, ::Complex{Int64})

```

实际上，这种比较在数学上也是无意义的。

2. 相等或相同

“小于”及“大于”这两种运算符是互斥的，不可能同时成立，但也有可能都不成立，因为还有第三种情况，即两者相等。我们可以使用等于(==)或者不等于(!=或≠)判断。

同样，“不等于”或“等于”这两者也是互斥的，不可能同时成立。例如：

```

julia> Int32(10) == Int64(10)              # 类型不同
true

julia> UInt8(5) == Int64(5)                # 类型不同
true

julia> 5 != 3
true

julia> Int32(15) != Int8(15)
false

julia> Int32(1) == Float64(1.0)            # 浮点型与整型对比时，可以获得正确的结论
true

```

可见，是否相等的比较会忽略值的具体类型，而直接对数值本身进行比较。

另外，与大小比较不同的是，是否相等对于有理数以及复数类型是适用的，例如：

```

julia> 15//8 == 3
false

julia> 15//8 != 1.875                      # 有理数15//8的实际值为1.875
false

julia> 3+1im == 3+im
true

julia> 3+2im == 1.875
false

julia> 15//8 != 3+2im
true

```

必要时，如果判断是否相等的同时需要考虑值的类型，则可用===这个“完全相同”



运算符，例如：

```
julia> Int32(10) === Int64(10)    # 类型不同
false

julia> Int32(15) === Int8(15)     # 类型不同
false

julia> 3+1im === 3+im              # 类型与值均相同
true

julia> Int32(10) === Int32(10)    # 类型与值均相同
true
```

如例中所示，该运算符只有在两个操作数的值与类型均相同时，才会返回 `true` 值。

运算符 `!==` 或 `≠` 则与之相对，表示“不全相同”，值或类型不一致便会返回 `true` 值，例如：

```
julia> Int32(19) === Int32(19)    # 类型与值均相同
true

julia> Int32(19) !== Int32(19)    # 类型与值均相同
false

julia> Int32(19) !== Int64(19)    # 类型不同
true

julia> Int32(19) !== Int64(10)    # 类型与值均不同
true
```

如果在开发中需要严格的比较，可以考虑使用该运算符。



注意 事实上，是否相等及是否相同可以适用于两个操作数是任意类型的情况，即可以对任意两个对象使用这两类操作符，判断它们是否相同或相等。但大小比较并非如此，需要操作数是同类型，而且该类型在数学上要有意义才行。这点在开发中是需要注意的。

3. 特殊值比较

上述的比较运算符对无穷值也是适用的，例如：

```
julia> Inf32 == Inf                # 不同类型的正无穷是相等的
true

julia> Inf == Inf
true

julia> Inf > Inf
false

julia> Inf < Inf
false
```

但对于 NaN 这个特殊的浮点值，比较结果却相当特别，如下：



```
julia> NaN == NaN
false

julia> NaN > NaN
false

julia> NaN < NaN
false

julia> NaN != NaN
true
```

无论是大于、小于还是等于均不能成立。

这一点可以从 NaN 的本质意义上去理解。NaN 设立的目的是表达一种无效值，而无效值本质上并非是一个具体的值，所以无从进行比较。另外，在将 Inf 或普通数值与 NaN 进行比较时，也遵循这个原则，例如：

```
julia> Inf > NaN
false

julia> Inf == NaN
false

julia> Inf < NaN
false

julia> NaN != Inf
true
```

实际上，在对无穷值与 NaN 在内的操作数进行比较时，Julia 遵循 IEEE 754 标准^①的约定：

- ❑ 正无穷大等于自身，但大于除了 NaN 外的任何数值。
- ❑ 负无穷大等于自身，但小于除了 NaN 外的任何数值。
- ❑ NaN 不等于、不小于也不大于任何数，包括自身。
- ❑ 正零等于但不大于负零。

但在实际使用中，往往希望将同为 NaN 的两个值作为同一事物进行处理，或者需要区分“正零”与“负零”，则可使用 `isequal()` 函数。通常情况下，该函数等效于 `==` 这个运算符，仅在值为 NaN 或零时会有所不同。例如：

```
julia> isequal(NaN, NaN)
true

julia> isequal(NaN, NaN32)
true

julia> -0.0 == 0.0
true
```

① IEEE 754 标准是二进位浮点算术标准（IEEE Standard for Floating-Point Arithmetic）的编号，等同于国际标准 ISO/IEC/IEEE 60559。该标准由美国电气电子工程师学会（IEEE）计算机学会旗下的微处理器标准委员会（Microprocessor Standards Committee, MSC）发布。




```
julia> isequal(-0.0, 0.0)
false
```

开发者可以根据应用的需求进行选择。

4. 其他比较操作

运算符“约等于” \approx 和“不约等于” \neq 是 Julia 有特色的部分。两者的用途相反，用于判断两个操作数是否在默认的阈值下满足大约相等关系。例如：

```
julia> 0.1  $\approx$  (0.1 - 1e-10)      # 两个操作数相差较小
true

julia> 1.3  $\approx$  1.300000002        #相差较小
true

julia> 1.3  $\neq$  1.300002          #相差较大
true
```

这两个运算符在不需要精确对比的时候能够提供不少便利。

另外两个操作符，“大于等于”(\geq 或 \geq)以及“小于等于”(\leq 或 \leq)，则可同时对上述的两种情况进行判断。但同样，这两类运算符不适用于复数。至于这两类运算符的用法，因与大小比较相似，不再赘述。

此外，对于多个操作数的比较，Julia 还提供了链式比较表达式 (Chaining Comparisons)，例如：

```
julia> 1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 < 3 != 5
true
```

不过由于可读性较差，笔者不建议使用，也不再过多介绍，有兴趣的读者可参见官方文档进一步了解。

4.5 逻辑运算符

比较运算实现了数值之间大小的判断，构成了程序最为常见的基本判断条件。但如果需要同时对多个条件进行综合判断，则需要使用逻辑运算符。

与比较运算符不同的是，输入的操作数不再是数值或返回数值类型的表达式，而是布尔值或返回值为布尔型的逻辑表达式。与比较运算符相同的是，逻辑运算符会在所有条件满足时返回 `true` 值，不满足时返回 `false` 值。

在 Julia 中，逻辑运算符有与、或、非三种，具体如表 4-4 所示。

表 4-4 逻辑运算符

运 算 符	说 明
&&	与此同时（二元）
	或者（二元）
!	不满足（一元）



例如, `a && b` 表示若条件 `a` 成立而且 `b` 也成立, 该表达式的结果为 `true` 值; 表达式 `a || b` 则意味着只需 `a` 或者 `b` 任一个成立, 便取 `true` 值。表达式 `!a` 则是反转 `a` 的真假, `a` 不成立时, 表达式 `!a` 的结果才为 `true` 值。

因为本就是为了对多个条件进行同时判断, 所以逻辑运算符自然地支持链式表达, 例如:

```
julia> a = true; b = false; c = true; d = false;
julia> a && b || c && !d
true
```

通常在控制流程中使用逻辑运算符, 以便在满足不同条件时执行不同的处理过程。

4.6 运算优先级

表达式中往往会同时使用多种运算符, 还会有其他的各种限定符、操作符 (例如有理数表达时的 `//` 符号, 为了叙述方便, 后文将所有可以进行计算或对计算进行控制的符号称为操作符)。在对这种表达式求值时, 需要各种运算符、操作符有明确的先后执行顺序, 即优先级。

Julia 中包括运算符在内的各种常见操作符的优先级如表 4-5 所示 (中文是其后操作符的分类说明)。

表 4-5 Julia 操作符优先级

优先等级	操作符列表
1	赋值 = 位运算 ~ 更新 += -= *= /= //= \= ^= ÷= %= = &= ⊔= <=> >>=>
2	键值 (Pair) ==>
3	条件运算 ? (三元运算符 ?: 中的首操作符)
4	逻辑
5	逻辑 && !
7	比较 == != ≠ === ≠ !== ≠ ≈ < <= > >= ≥ 类型断言 <: >:
8	语法操作符 <
9	语法操作符 >
10	语法操作符 :..
11	算术 + - 位运算 ⊔
12	位运算 >>> >> <<
13	算术 * / \ % ÷ 位运算 &
14	有理数 //
15	算术 ^
16	类型限定 ::
17	成员引用 .

表中的优先等级越大, 则对应的操作符计算优先级越高, 会优先执行。



Julia 中的操作符非常丰富，表中列出的有些我们还没有详细介绍，将在后面进行学习。



提示 如果要获得 Julia 操作符的完整列表，可以查看其源代码中的 `src/julia-parser.scm` 文件，其中还有大量的 Unicode 操作符。

对于任意的操作符，可以通过内置的 `Base.operator_precedence()` 函数查看其优先等级。但因为操作符的特殊性，所以将其作为参数时，需要使用标识符：表达（即以 `Symbol` 类型提供，后面介绍），必要时还需要使用圆括号进行界定以避免歧义。例如：

```
julia> Base.operator_precedence(:+)
11

julia> Base.operator_precedence(:+=)
1

julia> Base.operator_precedence(:(=))
1

julia> Base.operator_precedence(:(::))
16
```

如果提供的操作符无效，该函数会返回 0 值。需要注意的是，其中的 `Base` 是 Julia 的模块名（后面介绍），不能省略。

4.7 类型提升

本节因涉及尚未介绍的 Julia 类型系统、多态机制及其他内部原理，所以建议作为选读内容。

正如前文所述，在涉及多个类型互操作的过程中，比如累加或者其他算术运算，一般需要将 these 值先转换为相同的类型，以保证它们具有一致的内存结构，同时目标类型也会选择“较大”的类型^①。所谓“较大”，指的是该类型能够准确地容纳所有的操作数及计算结果，通常是这些操作数类型中表达能力最强的一个。例如浮点值 3.56 和整型值 8 在一起运算时，数值 8 会被提升为浮点型，因为整型没有小数部分，所以没有浮点型的表达力强。

这样的过程在 Julia 中称为“类型提升”（Promotion）。对于任意类型、数量的操作数，可以通过 `promote()` 函数将它们转为共同的类型。例如：

```
julia> promote(UInt8(1), Int32(1), UInt32(1))
(0x00000001, 0x00000001, 0x00000001)      # Tuple{UInt32, UInt32, UInt32}

julia> promote(Int32(1), Float32(1.0), Float16(1.0))
(1.0f0, 1.0f0, 1.0f0)                      # Tuple{Float32, Float32, Float32}

julia> promote(pi, 1//2)
(3.141592653589793, 0.5)                   # Tuple{Float64, Float64}

julia> promote(Int64(1), 1//2)
(1//1, 1//2)                               # Tuple{Rational{Int64}, Rational{Int64}}
```

^① 可通过内置的 `widen()` 函数获知某个类型的较大类型，例如 `widen(Int8)=Int32`。


```
julia> promote(1//2, Float32(1))
(0.5f0, 1.0f0)           # Tuple{Float32,Float32}

julia> promote(1+2im, Float64(1))
(1.0 + 2.0im, 1.0 + 0.0im) # Tuple{Complex{Float64},Complex{Float64}}
```

在这样的类型提升系统中，浮点数会被提升到更大的浮点类型；整型会被提升到更大的整型；整型与浮点型混合时，会被提升到足够大的浮点型以能容纳所有的值；整型与有理数混合时，会被提升到有理数类型；有理数与浮点数混合时，会被提升到浮点数；复数与实数混合时，会被提升到相近的复数类型。

可见提升过程并非像看起来那样是隐式、自动进行的，这是因为在 Julia 内部已经定义了大量的类型提升规则。在多类型混合计算时，Julia 只是遵循了这些规则实施而已。事实上，Julia 中有一种多态分发机制，能够根据参数列表的差异选择同名函数的不同实现方法。所以 Julia 对类型有着较为严格的控制，不会随意对类型进行自动转换，尤其是函数的输入参数。运算符是语法有些特别的函数对象，操作数便是其输入参数。所以运算符也遵循多态机制，不会自动进行参数类型的转换，所谓的自动提升只是因为存在对应规则的存在。

如果因故需要自定义类型提升规则（比如声明了新的类型），则可以通过 `promote_rule()` 函数实现，例如：

```
promote_rule{::Type{Float64}, ::Type{Float32}} = Float64
```

表示当 `Float64` 与 `Float32` 混合计算时，目标类型采用 `Float64` 类型。如果运算中遇到了两者作为操作数的情况，Julia 内部便会通过 `promote()` 函数依照上述规则，将操作数均转换为 `Float64` 类型，然后再进行计算得到最终的结果。在我们声明创建了自己的新类型时，可以通过上述这样的机制，将其对接到 Julia 的类型提升系统中。

因为 Julia 对类型的严格控制，所以若要针对各种具体类型重复地定义某个函数的实现方法，以多态的方式支持各种参数组合是极为麻烦的。不过，我们可以借助抽象类型的特点，并结合类型提升机制，简洁高效地实现这样的需求。例如：

```
+(x::Number, y::Number) = +(promote(x,y)...)
-(x::Number, y::Number) = -(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
/(x::Number, y::Number) = /(promote(x,y)...)

```

其中，四则运算为了能够适应任意两个数值类型，在参数类型中限定为抽象类型，这样可以接收整型、浮点型、复数等类型作为参数，并在函数实现体中通过类型提升机制将它们进行统一处理。

本节涉及的抽象类型、多态分发机制、函数定义等概念均会在后续章节详细介绍。此处不作赘述，如果读者感兴趣，可以学习完后续内容再回顾本节，便能够领略到这种方案的灵活强大之处。

控制逻辑

控制逻辑的作用是对语句的执行顺序进行更为灵活的控制，从而适应各种不同的实际需求。Julia 在程序逻辑控制方面的支持极为简洁，主要有判断分支与循环迭代两种，还有必要的异常处理流程。本章先从复合表达式开始逐步介绍这方面的内容。

5.1 复合表达式

一个完整的处理过程往往需要多个语句或表达式共同实现。这些相关表达式构成的独立语句块称为复合表达式 (Compound Expressions)，而其内部的表达式称为子表达式。

Julia 中有两种方式构造复合表达式，其中一种使用 `begin` 与 `end` 这对关键字，如下所示：

```
begin
    # 任意多的子表达式
end
```

另一种则使用分号将多个子表达式连接，以链式的方式构造，如下所示：

```
子表达式; 子表达式; ....
```

在必要时，可用圆括号将子表达式包括起来，以免语法歧义，如下所示：

```
(子表达式; 子表达式; ....)
```

复合表达式会将最后一个子表达式的计算结果作为整体的返回值，例如：

```
julia> z = begin
           x = 1
           y = 2
           x + y    # 即 1+2
       end
```

或者:

```
julia> z = x = 1; y = 2; x + y
3

julia> z = (x = 1; y = 2; x + y)
3
```

当然在书写中并不限定 begin 方式一定要多行, 例如:

```
julia> begin x = 1; y = 2; x + y end
3
```

Julia 并未要求将分号作为语句的结尾字符, 但如果多个表达式在一行中书写, 则需使用分号隔开, 所以上例中的分号不可省略。

另外, 链式表达中也并不限定必须是单行, 例如:

```
julia> (x = 1;
        y = 2;
        x + y)
3
```

可见复合表达式的表述方法非常灵活。


5.2 判断逻辑

在实际的业务逻辑中, 常需根据不同的情况作出判断, 并选择对应的分支以执行不同的处理过程, 让程序适应场景的变化。这是判断逻辑 (Conditional Evaluation) 的经典功能。

在 Julia 中, 关键字 if、elseif、else 和 end 组成了判断逻辑的主要结构, 基本语法为:

```
if 条件表达式1
    # 实现体
elseif 条件表达式2
    # 实现体
elseif 条件表达式3
    # 实现体
else
    # 实现体
end
```

其中, “条件表达式” 不需要像 C++ 语言那样使用括号界定, 只需用空格与 if、elseif 这两个关键字隔开即可。不过, 条件表达式一定要是能够返回布尔型结果的逻辑运算。

 **提示** 结构中的 if 是必需的, 且只能出现一次; else 可以不出现但若出现则只能一次; 而 elseif 可多次使用, 但需在 if 之后、else 之前。

实际上, 每个关键字 (除 end 之外) 之后的语句块均可视为复合表达式, 都可以有自己的返回值, 例如:

```
julia> if 3 > 0 || 2 > 0 && !(0 > 3)
        2
else
    end
```

66 ◆ Julia 语言程序设计

```

        -2
    end
2

julia> if x > 0
    "positive!"
else
    "negative..."
end
"positive!"

julia> z = if 3 < 0
    10
else
    -10
end
-10

```

Julia 中的数值本身同样可以构成独立的语句，也同样是表达式，所以也是其所处复合表达式的子表达式。当一个数值或变量处于复合表达式的最后时（如上例所示），其便是该复合表达式的返回值。

在判断逻辑的运行过程中，程序会按需对 if 与 elseif 的逻辑条件进行判断，如果表达式为 true，则会进入对应的语句块执行，执行完成后便会直接返回其中复合表达式的结果，不再执行结构中的其他语句，即 if 或 elseif 构成的多个语句块中只有一个会被执行。例如：

```

julia> if 3 > 0          # 条件为true,
    3                    # 运行返回
elseif 2 > 0            # 条件为true,
    2                    # 但不会运行
end
3

julia> if 0 > 3          # 条件为false,
    0                    # 不会运行
elseif 3 > 0            # 条件为true,
    3                    # 运行返回
elseif 2 > 0            # 条件为true,
    2                    # 但不会运行
else
    nothing
end
3

```

需要注意的是，在为 if 或 elseif 提供条件表达式时，是不支持将 1 或 0 这种整型值直接作为 true 或 false 使用的，若那样做会直接报错，例如：

```

julia> if 1
    println("true")
end
ERROR: TypeError: non-boolean (Int64) used in boolean context

julia> if !0
    println("this is error")
end
ERROR: MethodError: no method matching !(::Int64)

```

在 Julia 中, 1 或 0 与 true 或 false 是完全不同的类型, 不能混用, 下文介绍的循环结构中同样如此。

对于语句简单或表达式并不复杂的情况, 可使用三元运算符?: 实现判断逻辑。例如, 在表达式 `a ? b : c` (注意各符号之间有空格) 中 `a` 是条件表达式, 值为 true 时该三元运算符返回 `b` 值, 否则返回 `c` 值。例如:

```
julia> x=3; x>10 ? true : false    # x为3, 所以>10不成立, 返回第二个值false
false

julia> 3>0 ? 3 : 0
3

julia> false ? 3 : 0
0
```

需要注意的是, 在 v1.0 版的 Julia 中, 符号?及:的前后都需要有空格, 否则会报错。

此外, 这种简单的逻辑判断, 还可以考虑使用内置的 `ifelse()` 函数, 其原型为:

```
ifelse(condition::Bool, x, y)
```

当 `condition` 为 true 时返回 `x`, 否则返回 `y` 值。例如:

```
julia> ifelse(1>0,1,0)
1

julia> ifelse(1<0,1,0)
0
```

函数 `ifelse()` 可实现 `if` 结构或?: 类似的功能, 但本质却是不同的。很多时候使用 `ifelse()` 函数代替 `if` 结构能够消除过多的逻辑分支, 生成更为高效的代码。

5.3 循环逻辑

不同于判断逻辑, 循环逻辑 (Repeated Evaluation) 会在条件满足的情况下, 重复地运行一段代码以实现数据迭代处理等功能。

在实现循环逻辑时, 我们在考虑什么情况下需要重复执行的同时, 也需要考虑什么情况下不再循环。这是在开发中不能忽略的两个对偶问题。

Julia 提供了 `while` 与 `for` 这两种循环结构的同时, 还提供了 `break` 与 `continue` 两个关键字, 以便对循环进行控制。通过设定与业务功能相关的控制变量, 并结合这两个关键字, 便可实现在特定情况下终止循环或继续循环。

5.3.1 while

基于关键字 `while` 的循环结构同 `if` 结构类似, 也需要将逻辑表达式作为判断条件, 基本语法如下:

```
while 条件表达式
```



```
# 语句块
end
```

只有在“条件表达式”为 true 时，while 内部的语句块才会执行。

在 while 内部的语句运行到 end 时，会再次返回对条件进行判断，如果为 false，则忽略 while 内部的语句块，直接跳到 end 处执行其后的其他语句。

为了对 while 循环进行动态控制，一般其判断表达式中有控制变量，而且能在内部语句块进行更新，从而可以修改下次判断条件的逻辑结果。例如：

```
julia> i = 1;           # 循环控制变量
julia> while i <= 5     # 循环条件
    print(i, " ")      # 处理过程
    global i += 1       # 改变控制变量的值，其中的global关键字后文会介绍
end
1 2 3 4 5
```

若非如此，条件便始终不变，while 就会变成死循环或无限循环。



提示 只有在必要的情况下才会采用无限循环，例如在服务器接受远程客户端连接的代码中。为了能够提供全时候的服务，服务端一般会以死循环的方式等待并不断接受客户端进入。

当然，对于条件表达式结果恒定不变的无限循环，也可以在满足内部某条件时通过关键字 break 打断循环过程。例如：

```
julia> i = 1;           # 控制变量
julia> while true       # 始终满足条件
    print(i, " ")      # 处理过程
    if i > 5            # 循环控制
        break          # 满足控制条件即会强制退出循环
    end
    global i += 1       # 修改控制变量
end
1 2 3 4 5 6
```

其中，虽然 while 的条件表达式始终为 true，成为无限循环结构，但通过控制变量与 break 的结合使用，可在需要时退出循环。此例实现了与上例相似的功能，只是结果略有不同，有兴趣的读者可以对比研究一下。

关键字 break 能够强制循环退出，但在多层嵌套的循环中只会影响其所处的那一层，不会波及外层或内层循环结构。例如：

```
julia> i=1; j=1;
julia> while j <= 4
    while true
        print(i, " ")
        if i > 5      # 内层循环控制
            break     # 强制退出内层循环
        end
        global i += 1 # 修改内层循环的控制变量
    end
    global j += 1     # 修改外层循环的控制变量
    global i = 1      # 重置内层循环的控制变量
    println()         # 换行打印
```

```

end
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6

```

例中 `break` 导致内层 `while` 循环只执行到 `i=6` 时跳出，但外层由 `j` 控制的 `while` 循环并没有受到 `break` 的任何影响，顺利执行了 4 次。

同样用于循环控制的 `continue` 关键字，功能不同于 `break`，不是打断循环，而是让循环运行到其位置时停止执行后续语句，并立即跳转到条件表达式处重新判断，如果条件成立便会再次重入循环。例如：

```

julia> i = 1;
julia> while i < 10
    global i += 1      # 修改控制变量
    if i < 5           # 循环控制
        continue      # 满足i<5时跳到while处
    end
    print(i, " ")     # 处理过程
end
5 6 7 8 9 10

```

其中，循环控制变量 `i` 以 10 为上界，只要小于 10 便会不断运行循环体内的语句；但在 `i` 小于 5 的情况下，不会执行 `continue` 之后的代码，所以打印出的结果都是不小于 5 的数值。

类似于 `break`，关键字 `continue` 也只控制着其所处的当前层循环，不影响外层或内层的循环结构。例如：

```

julia> i=1; j=1;

julia> while j <= 4
    while i < 10
        global i += 1      # 修改内层循环的控制变量
        if i < 5           # 内层循环控制
            continue      # 忽略内层循环后续语句
        end
        print(i, " ")
    end
    global j += 1          # 修改外层循环的控制变量
    global i = 1          # 重置内层循环的控制变量
    println()             # 换行打印
end
5 6 7 8 9 10
5 6 7 8 9 10
5 6 7 8 9 10
5 6 7 8 9 10

```

这些细节需要在开发过程中仔细规划，以免引入不必要的逻辑问题。

5.3.2 for

另外一种循环由关键字 `for` 构建，与 `while` 的用法有很大的不同。在 `for` 循环中，

控制循环的不再是条件表达式，而是遍历型表达式。其基本表述结构为：

```
for 遍历表达式
    # 语句块
end
```

其中，“遍历表达式”需要可迭代数据结构的支持。

一般而言，在数集迭代时便会驱动循环体的语句块一次次地执行；而循环的次数便是数集中可遍历到的元素个数。同时，遍历表达式本身会提供循环变量，用于逐一提取数集中的元素值，以便循环内部进行相应的处理。

Julia 中提供了三种基本的遍历表达方式，如下：

```
for 循环变量 = 可迭代数据集
    # 更多语句
end
```

```
for 循环变量 in 可迭代数据集
    # 更多语句
end
```

```
for 循环变量 ∈ 可迭代数据集
    # 更多语句
end
```

其中，三个遍历操作符 =、in 及 ∈ 是等价的，可以任选一个。

所谓可迭代数据集，主要是指后面介绍的元组 (Tuple)、字典 (Dict) 及集合 (Set)，或者数组 (Array) 及范围表达式 (Range) 等。为了叙述方便，这里作简单的说明：元组的形式为 (1, 2, 3)，一旦创建内部元素不能再变更；数组是多维数据序列，以中括号表达，例如 [1, 2, 3] 或 [1 2 3] 便是有三个元素的数组；而范围表达式则用于表达等差序列，是一种迭代器，例如 1:10 表示公差为 1 的整数序列，取值 1 到 10 共计 10 个元素。

下面我们以实例来了解 for 循环的使用，例如：

```
julia> for i = 1:5
    print(i, " ")
end
```

```
1 2 3 4 5
```

```
julia> for i in 1:5
    print(i, " ")
end
```

```
1 2 3 4 5
```

```
julia> for i ∈ 1:5
    print(i, " ")
end
```

```
1 2 3 4 5
```

或者：

```
julia> for i = [1 2 3 4 5]
    print(i, " ")
end
```

```
1 2 3 4 5
```

```
julia> for i in [1 2 3 4 5]
    print(i, " ")
end
```

```
1 2 3 4 5
```

```
julia> for i ∈ [1 2 3 4 5]
    print(i, " ")
end
```

```
1 2 3 4 5
```

又或者:

```
julia> for i = (1,2,3,4,5)
    print(i, " ")
end
```

```
1 2 3 4 5
```

```
julia> for i in (1,2,3,4,5)
    print(i, " ")
end
```

```
1 2 3 4 5
```

```
julia> for i ∈ (1,2,3,4,5)
    print(i, " ")
end
```

```
1 2 3 4 5
```

在 for 循环中, 还支持同时对多个数据集进行遍历, 例如:

```
julia> for i = 1:2, j = (1,2,3,4,5)
    print((i,j), " ")
end
```

```
(1, 1) (1, 2) (1, 3) (1, 4) (1, 5) (2, 1) (2, 2) (2, 3) (2, 4) (2, 5)
```

而且遍历操作符也可以混用, 例如:

```
julia> for i = [1 2 3 4 5], j in 3:4
    print((i,j), " ")
end
```

```
(1, 3) (1, 4) (2, 3) (2, 4) (3, 3) (3, 4) (4, 3) (4, 4) (5, 3) (5, 4)
```

在这种多遍历的表达中, 得到的循环变量组合实际是每个数据集的笛卡尔积。也正因如此, 遍历操作的顺序影响着内部语句中循环变量的取值。如果更换遍历操作的顺序, 将会得到不同的结果。例如:

```
julia> for j = (1,2,3,4,5), i=1:2
    print((i,j), " ")
end
```

```
(1, 1) (2, 1) (1, 2) (2, 2) (1, 3) (2, 3) (1, 4) (2, 4) (1, 5) (2, 5)
```

```
julia> for j in 3:4, i = [1 2 3 4 5]
    print((i,j), " ")
end
```

```
(1, 3) (2, 3) (3, 3) (4, 3) (5, 3) (1, 4) (2, 4) (3, 4) (4, 4) (5, 4)
```


可见与上述的运行结果不同。其中的 i 或 j 虽然仍出自同一数集，但因为遍历顺序的不同，循环变量的组合出现了差异。

还有一点需要注意，每一个遍历表达式之间的分割符是逗号而非是分号。如果使用分号，则会产生出乎意料的结果。例如：

```
julia> for i=1:2; j=3:4
    print((i,j), " ")
end
(1, 3:4) (2, 3:4)
```

可见只执行了 2 次循环，而不是 4 次；而且 j 的值不再是元素，也没有实现笛卡尔积。这是因为 $i=1:2$ ； $j=3:4$ 这种表达不再是遍历表达式，而是复合表达式。

当然，在 `for` 循环中，我们同样可以使用 `break` 与 `continue` 来控制循环的运行。例如：

```
julia> for i = 1:1000
    print(i, " ")
    if i >= 5
        break
    end
end
1 2 3 4 5
```

例中的 i 在遍历表达中可以取到 1000 个数值，但因内部的 `if` 语句要求 $i \geq 5$ 时 `break`（中断循环），所以该循环只实际执行了 5 次。再例如：

```
julia> for i = 1:10
    if i % 3 != 0
        continue           # 重新跳到 for i = 1:10
    end
    println(i)
end
3
6
9
```

即只有在 i 是 3 的倍数的时候，才会将 i 的值打印出来。

另外，在 `for` 循环中，`continue` 的运作与在 `while` 中的表现是相同的，也同样只会影响其所处的内层循环，不会波及外层循环。例如：

```
julia> for j = 1:5
    println("\nloop ", j) # \n为换行符，便于查看结果
    for i = 1:10
        if i % 3 != 0
            continue       # 重新跳到 for i = 1:10
        end
        print(i, " ")
    end
end

loop 1
3 6 9
loop 2
```

```

3 6 9
loop 3
3 6 9
loop 4
3 6 9
loop 5
3 6 9

```

其中由 `j` 控制的外层循环完整地运行了 5 次，而由 `i` 控制的内层循环只在 `i` 为 3 的倍数时完整地运行了 3 次，但实际仍运行了 10 次。

5.4 异常处理

由于输入的不确定性、运行环境的变更、交互的复杂性或设计的缺陷，程序总会存在一些漏洞 (Bugs)。可以说没有不存在 Bug 的程序。

很多情况下 Bug 在发布前是难以发现的，都是在实际使用中不断地出现再不断地修复，但在这不断优化改善的过程中又有可能引入新的漏洞或缺陷。这是软件工程常见的客观情况。为此，在必要的时候，对一些代码采取预防措施能够适当地减少异常引发的程序崩溃。所以，异常处理机制也是很多语言的标配。



提示 本节为了展示异常机制，会涉及函数的定义和调用，其中的不少概念会在后续章节介绍，如果存在阅读难度，可在学习第 7 章“函数”后再回顾本节内容。

5.4.1 异常触发

由于程序设计时，每段代码的正确执行都需要特定的前置条件，而通常这些条件是可知。一旦条件不满足，会导致后续主要逻辑的不正常或无法运行，为此可以对这些可预知的“错误”进行提前设定处置逻辑，这些分支逻辑即是异常处理逻辑。

另外，既然是可知的错误，就可以预先对某些常见的错误进行定义，以便于后续的跟踪与处置。Julia 中内置定义了很多错误类型，均继承自抽象类型 `Exception`。本书在附录 A 中列出了常见的异常类型，更多的可参见官方文档。

对于这些预定义的错误，可在需要的时候通过 `throw()` 函数显式地抛出，以上报该处代码发生的问题，并方便后续的调试与处置。

例如有一个函数只能处理非负数，却收到了负数参数，如下所示：

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError("x<0")) # 当x<0时抛出DomainError异常
f (generic function with 1 method)
```

```
julia> f(1)
0.36787944117144233
```

```
julia> f(-1)
ERROR: DomainError with x<0:
```

负数作为参数

```
Stacktrace:
```

```
[1] f(::Int64) at .\REPL[1]:1
[2] top-level scope at none:0
```

可见一旦运行条件不符合要求，DomainError 的实例便会被抛出，同时会传递关于异常的描述信息。



提示 在异常信息之后，同时会有 Stacktrace 字段给出错误发生的具体函数原型及其定义的位置。因为该字段主要提示错误发生的位置，所以每次原型都会有所差异，本书因为篇幅，在下面的示例中出现错误时会省略该字段，仅展示 ERROR 信息。

除了预定义的错误类型，还有一个通用的异常类型 `ErrorException`，能够在上报时以简短的信息描述错误的情况。若是希望自定义异常类型，则可声明 `Exception` 新的子类型，此后便可通过函数 `throw()` 在需要的时候上报新声明的错误类型，以便进行恰当的处置。

若无须针对错误的类型进行后续处理，而仅需对错误进行提示，则可直接通过 `@error` 宏上报。例如：

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : @error "negative x not allowed"
fussy_sqrt (generic function with 1 method)

julia> fussy_sqrt(2)
1.4142135623730951

julia> fussy_sqrt(-1)
└ Error: negative x not allowed          # 在REPL中Error会以红色显示
└ @ Main REPL[36]:1
```

这其实并非属于异常类型机制中的内容，而是开发中最为常用的日志工具，也是发现问题最为简单有效的手段。

与 `@error` 类似的还有 `@info`，`@warn` 和 `@debug`，分属于不同的日志级别，依次为常规信息、警告信息及调试信息。当然，它们打印的信息前缀是有所区别的，例如：

```
julia> @info "information message"
[ Info: information message

julia> @warn "warning message"
└ Warning: warning message
└ @ Main REPL[40]:1
```

关于日志方面，Julia 有一个 `Logging` 包，提供了更为全面的功能，开发者可以根据需要安装使用。通过在代码恰当的位置输出对应的信息，能够在程序的编写、调试及测试方面带来很大的帮助。

5.4.2 异常捕捉

在异常发生时，`throw()` 抛出错误后，需要对其进行捕捉才能在恰当的位置有针对性地进行处置。在 Julia 提供的捕捉机制中，将可疑语句放在 `try` 和 `end` 组成的异常处理结构中，如下所示：

```

try
    # 正常语句块
catch 异常类型变量
    # 异常处理语句块
finally
    # 收尾清理语句块
end

```

其中, `try` 和 `end` 是必需的, 两者之间的语句块便是预期正常运行的业务代码, 可视为复合表达式; `catch` 是异常捕捉语句, 其后一般会跟着一个“异常类型变量”, 用于提取 `try` 中发生之的异常类型, 可针对异常类型变量的不同情况进行异常处置; 而 `finally` 中的语句块用于清理收尾, 该部分的代码无论是否发生异常, 都会执行。

在异常捕捉结构中, 除了 `try` 和 `end` 必需外, `catch` 和 `finally` 两者也必须至少提供一个。下面给出一个的例子:

```

julia> try
    log(-10)
catch ex
    tuprintln("exception type: ", ex)
finally
    println("here is finally")
end
exception type: DomainError(:log, "-10.0 will only return a complex result if
called with a complex argument. Try -10.0(Complex{x}).")
here is finally

```

其中, `log()` 函数要求参数必须是正数, 但提供的参数却是负数, 所以会报错。运行中, 异常发生时, 类型 `DomainError()` 被记录在了变量 `ex` 中。再例如:

```

julia> except_test(x) = try
    if 1 == x
        log(-10)
    else
        x[2]
    end
catch ex
    if isa(ex, DomainError) # 针对不同的类型做不同的处理
        println("Caught DomainError")
    elseif isa(ex, BoundsError)
        println("Caught BoundsError")
    end
end
except_test (generic function with 1 method)

julia> except_test(1)
Caught DomainError

julia> except_test(0)
Caught BoundsError

```

在异常处理的语法结构中, `finally` 在对文件、网络等资源操作时比较有用。因为无论 `try` 代码块是否正常执行, 还是 `catch` 代码块是否恰当地处理了异常, `finally` 中的代码块都会执行, 所以常用来关闭并释放各种资源, 进行各种清理工作。例如:

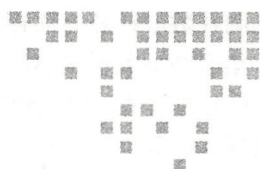

```
julia> f = open("file");

julia> try
    # 对文件对象f进行处理
catch ex
    # 异常处理
finally
    close(f)    # 关闭文件对象
end
```

上例中，无论 `try` 是否顺利地对 `f` 进行了处理，`close(f)` 都会执行。期间，若确实发生了异常，则先执行 `catch` 中的逻辑，然后也会自动流转到 `finally` 中。

在 Julia 的异常处理语法中，如果出现无法正确处理的错误，则可以通过 `rethrow()` 函数、`backtrace()` 函数或 `catch_backtrace()` 函数对发现的异常再次传播，留给外部的代码进行处置，或者通过嵌套的 `try` 语法结构进行更深层的处理。

虽然异常机制为代码的健壮性提供了一层保障，但异常处理机制不能当成一种特效药，更不能成为依赖。最好的防范措施是充分地测试，避免可能错误的出现。



类型系统

类型 (Types) 是语言中最为基本的要素, 用于描述数据的表达结构。但 Julia 的类型系统是非常独特的, 有着更为抽象、更为高层的语义。在该类型系统中, 不仅仅数值有类型, 函数、表达式, 甚至是类型本身与代码符号都有特定的类型。可以说, 一切语言要素在 Julia 中都对应着某个类型, 都是类型系统的一部分。在这种广泛、统一、完备、动态的类型系统下, Julia 的各种语言要素都是程序中可操纵的对象。

正是基于强大的类型系统, Julia 提供了对面向对象、函数式及泛型编程等多种范式的支持, 不但能够操作函数对象, 对类型进行参数化, 也能够进行多态分发 (Multiple Dispatch), 使得开发更为灵活、适应性更强。开发者可以宽松地使用类型, 快速地实现逻辑; 也可以在必要的时候严格地使用明确的类型, 获得更好的性能。

在第 3 章“数值系统”中已经介绍了 Julia 中的数值基本类型, 本章将介绍类型系统, 如抽象类型 (Abstract Types)、元类型 (Primitive Types)、复合类型 (Composite Types) 以及类型的参数化等内容, 之后还会介绍一些常用数集的基本使用方法。

6.1 类型简介

众所周知, 整数与小数是基本的数集, 有理数集是整数衍生出来的, 而无理数集是小数的子集, 它们都是实数集的一部分。与实数相对的是虚数, 也即复数集合。不过通常而言, 实数只是复数的一种特殊情况, 也可视为复数的子集。但不论是实数还是复数, 都是一种数字 (Number)。

不同的数集之间可以通过一些计算进行转换。例如两个整数的比值可获得有理数, 也可能获得无理数; 整数相除会获得小数, 而小数之间的加减乘除可得到整数; 或对负实数开方可获得复数, 等等。数集存在层层包含关系, 一层层地抽象为范围更大的集合, 基本



依据便是子集之间在数学计算方面存在着相容性与一致性。

显而易见，集合的抽象层次越高，内部元素就越具多样性。说到整数，很明确是 -1 、 1 、 3 等这种与数量有关的数值；但说到实数，因为其包含整数、小数、有理数、无理数等各种情况，只有再具体些，我们才能更准确地给出实例。

可以说，在数学理论中，实数是一种抽象概念，是抽象的数字类型；而整数是一种具体概念，是具体的数字类型。不过到了计算机领域，情况会变得更复杂。就目前的架构原理，计算机内部表达数字总会受到字节数的限制，所以其能够表达的数字形式是有限的。

一般而言，计算机中基本的数字同样是整数与小数，但为了与数学概念区分，分别称为整型与浮点型，都是数值（Value）的一种。遵照约定的编码规则表达某一数值时，可采用的字节或位（Bit）会有若干选择，也就有了多种存储结构（Storage Layout），例如前面介绍的单字节（8 比特位）整型 `Int8`、64 位整型 `Int64`、32 位浮点型 `Float32` 等。

在 Julia 中，这种与明确二进制存储结构相关联的数值类型被称为元类型，是可以在内存中分配空间并进行各种操作的具体类型；而内存分配产生的每一份值空间占用称为对应类型的实例（Instance）或对象（Object），这种通过空间分配创建对象的过程称为该类型的实例化（Instantiation）。

6.2 抽象类型

如上面所述，Julia 表达整数有多种不同结构，即 `Int8`、`UInt8`、`Int64`、`UInt64` 等，都是整型的一种。而整型一词便是这些具体整数类型的抽象，是一种统一性的描述。浮点型、实数型等同理。

作为以数值计算为特色的 Julia 语言，为了描述方便，建立与数学概念对等的概念，在语言层面为整型这种抽象概念提供了具象化的机制。我们可以通过 Julia 的语法规则明确地定义出类似于整型、浮点型、实数型，甚至是数字型等抽象类型。

在 Julia 中，抽象类型使用 `abstract type` 关键字声明，一般语法为：

```
abstract type name end
```

其中，`name` 为定义的抽象类型名称。例如：

```
abstract type Number end
```

便定义了一个名为 `Number` 的抽象类型，表示某个数据为数字。

不过在现实中，概念的抽象是可以延伸的，也会有很多层次，比如整数型到实数型，实数型到数字型，等等。为了表达这种抽象的层次结构，表述子集与超集关系，Julia 提供了标识符 `<`：用于说明类型之间的包含与继承关系，而且这种关系可以在声明类型直接提供，即：

```
abstract type name <: supertype end
```

该声明规则中的 `name` 的上一级抽象类型是 `supertype` 类型。可以说，`supertype` 的概念比 `name` 的层次更高，是 `name` 的超集，所以也称其为 `name` 的父类型；而 `name` 是



supertype 的子集,称为 supertype 的子类型。两者之间存在着明确的类型继承关系。例如:

```
abstract type Real <: Number end
```

表示实数 Real 类型是 Number 类型的子类型,是数字的一种。

如前所述,抽象类型只是描述同类具体类型的抽象概念,是一种语言机制,并没有明确的二进制内存结构与之关联,所以无法实例化,不能定义出拥有实际资源的对象。不过因为抽象类型实际是具体类型的超集,所以仍可在其上定义各种运算和操作,而且这些操作能够被其子类型继承复用。这其实与数学中空间域的性质是相通的,可以相互借鉴方便理解。

实际上,Julia 在数值方面已经内置定义了较为完整的抽象类型,部分如下所示:

```
abstract type Number end
abstract type Real <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer <: Real end
abstract type Signed <: Integer end
abstract type Unsigned <: Integer end
```

其中,除了 Number 与 Real 之外,还定义了前文未提及的 AbstractFloat 类型,用于描述所有的浮点类型;还有表达整数类型的 Integer 类型,并且分为 Signed 与 Unsigned 两种,分别指有符号整型及无符号整型;而 Integer 及 AbstractFloat 都是 Real 的子类型。

另外,Julia 还内置了特殊的抽象类型 Any,即“任意类型”,位于类型继承关系的顶部,覆盖了 Julia 中声明定义的所有类型,是所有类型的父类型,所以 Any 也是 Number 的父类型。

如果我们将包括 Any 在内的各种数字抽象类型的继承关系绘制成图,便会得到如图 6-1 所示的树状结构。

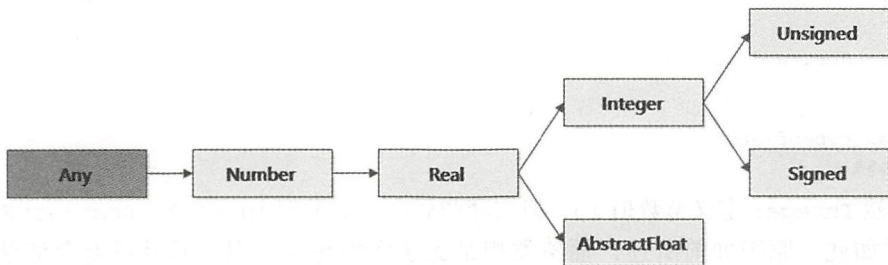


图 6-1 数字抽象类型拓扑树

从抽象类型的声明方式及图中,我们显然能够发现这种类型继承树是一种单继承树。在这棵树中,每个类型的上一级抽象类型只有一个,即有且只有父类型节点;同时,除了叶子类型外,其他都会有子类型,而且某些会有多个子类型,例如 Real 类型下至少有



`AbstractFloat` 及 `Integer` 两种；而同属于某一层父类型的各子类型称为兄弟类型。这种依据 Julia 类型声明规则形成的层次性继承关系树，构成了 Julia 类型系统的重要基础——类型拓扑图（Type Graph）。

在这样的拓扑结构中，节点中的任一个类型向上追溯，只会找到唯一的路径能最终到达顶层的 `Any` 节点。而且，处于同一路径上的任两个类型都存在父子继承关系。

在图 6-1 的类型拓扑树中，虽然 `Real` 不是 `Signed` 的直接父节点，但仍可称 `Real` 是 `Signed` 的父类型，或称 `Signed` 是 `Real` 的子类型，对于其他类型关系亦是如此。如此一来，如果我们在 `Real` 层面定义了某个操作或运算，会被其下的所有子类型所遵循。但如果某个运算定义在 `Integer` 级别，便不能被“旁路”的 `AbstractFloat` 等类型遵循，因为它们之间不存在父子继承关系，没有与 `Integer` 处于到 `Any` 节点的同一路径上。兄弟类型共享的只能是共同父类型上定义的计算规则，这也是 Julia 中强大的多态分发机制的基础。

类型间的继承关系在 C++、Java 语言中也广泛存在。但 C++ 中的多继承很容易被滥用，不但增加了维护难度，降低运行性能，还很容易引发其他各种不可预期的问题。Julia 采用单继承方式，正是吸取了这方面的教训。也正因为如此，Julia 的类型系统有着更为强大的生命力，不但更为实用、直观、清晰，而且简洁、更容易表达，为其他各种强大的功能奠定了坚实的基础。

6.3 元类型

如果我们试图以上述的抽象类型创建对象，会发现：

```
julia> Integer(10)
10

julia> typeof(ans)
Int64

julia> AbstractFloat(10)
10.0

julia> typeof(ans)
Float64
```

其中，虽然 `Integer` 定义整数值 10，但实际归属在具体类型 `Int64` 下；`AbstractFloat` 的情况同样如此。原因如前所述，抽象类型是无法实例化的，因为其并没有存储结构方面的定义。若要存储这些真实的数值，只能转为其子类型中的某个具体类型（这个过程会由 Julia 自动执行）。

所谓“元类型”，是具体类型的一种，不但有明确的二进制内存结构，也是构成其他数据的基础。简单来说，元类型是概念单一、结构纯粹的基本类型，其数据在内存中是连续的位序列。



所以在 Julia 中基于关键字 `primitive type` 声明一个元类型时，需要给出明确的位数：

```
primitive type name bits end
primitive type name <: supertype bits end
```

其中除了告知待声明元类型的名字 `name` 外，还需使用 `bits` 告知表达该类型时需要的位数，即该类型在容纳数据时的内存结构。

同抽象类型类似，在声明元类型时也可以同时给出其父类型。例如：

```
primitive type Float16 <: AbstractFloat 16 end
```

其中，声明了名为 `Float16` 的元类型，是浮点型 `AbstractFloat` 的一种，用 16 位表达浮点数。

实际上，第 3 章介绍的各种整型、浮点型等数字类型都是元类型的一种，都有着对应的元类型声明语句，如下所示：

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end
primitive type Bool <: Integer 8 end
primitive type Char <: AbstractChar 32 end
primitive type Int8 <: Signed 8 end
primitive type UInt8 <: Unsigned 8 end
primitive type Int16 <: Signed 16 end
primitive type UInt16 <: Unsigned 16 end
primitive type Int32 <: Signed 32 end
primitive type UInt32 <: Unsigned 32 end
primitive type Int64 <: Signed 64 end
primitive type UInt64 <: Unsigned 64 end
primitive type Int128 <: Signed 128 end
primitive type UInt128 <: Unsigned 128 end
```

其中，`Int8`、`Int16`、`Int32`、`Int64` 及 `Int128` 同是 `Signed` 的子类型，都是有符号整型；`UInt8`、`UInt16`、`UInt32`、`UInt64` 及 `UInt128` 同是 `Unsigned` 的子类型，都是无符号整型；而 `Float16`、`Float32` 及 `Float64` 则是浮点型 `AbstractFloat` 的子类型。

如果将这些元类型及上节介绍的抽象类型之间的继承关系绘制出来，便能够获得更为完整的数值类型拓扑图，如图 6-2 所示。

需要注意的是，在实际内存操作，一位作为独立的内存操作单元在机器指令中一般是不支持的。所以在定义元类型时，提供的位数值必须是 8 的倍数，即以字节为单位。上例中的布尔型是 `Integer` 的直接子类型，虽然理论上单比特就能够表达一个布尔值，例如 1 表示真，0 表示假，但仍以 8 位定义了存储结构。

另外，虽然布尔型、`Int8` 类型和 `UInt8` 在声明表述上一致，却并不能相互交换或替代，这是因为 Julia 采用的是主格（Nominative）类型[⊖]规则，而且它们有着不同的直接父类

⊖ 主格类型系统，即基于名称的类型系统，以声明的类型名称作为相容或等价判定的依据。相对的是结构类型系统，以内部结构作为判定依据，而不是名称。



型: Bool 的父类型是 Integer, Int8 的父类型是 Signed, 而 UInt8 的父类型则是 Unsigned 类型。

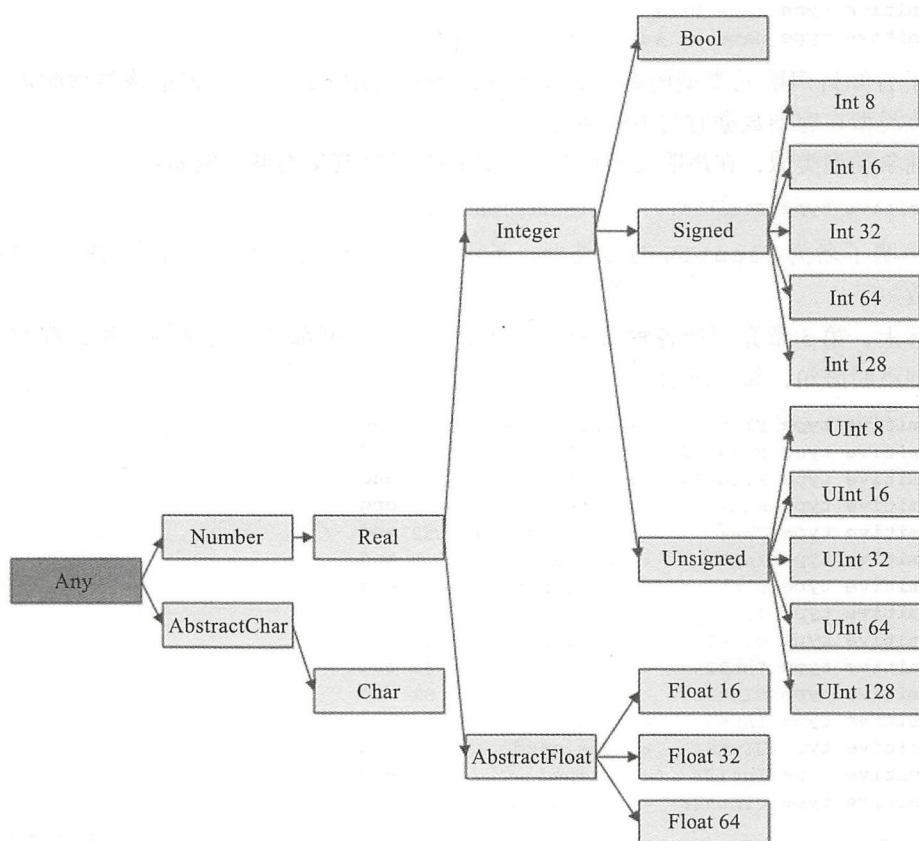


图 6-2 数字元类型拓扑树

在实践中, 如果要查询某个类型的父类型, 可使用 `supertype()` 函数, 例如:

```
julia> supertype(Signed)
Integer
```

```
julia> supertype(Integer)
Real
```

```
julia> supertype(Real)
Number
```

```
julia> supertype(Number)
Any
```

也可以使用 `subtypes()` 获得某个类型的子类型列表, 例如:

```
julia> subtypes(Signed)
5-element Array{Union{DataType, UnionAll},1}:
Union{Int8, Int16, Int32, Int64, Int128}
```




```
Int128
Int16
Int32
Int64
Int8

julia> subtypes(Real)
4-element Array{Any,1}:
  AbstractFloat
  AbstractIrrational
  Integer
  Rational
```

但是，该函数只会列出给定类型的直接后继子类型，不会进行递归查询。

6.4 类型操作

6.4.1 弱类型机制

在 Julia 中，类型与具体的值是密切相关的。值是占据内存的对象实体，是计算操作的真正目标，所以必然对应一个确切的类型，而且是可实例化的具体类型。即使是后续介绍的复合类型，其成员基础也是具体类型。

在类型与值之间，还有另外一个概念：变量。变量在 Julia 中并不像其他语言那样有着很重要的地位，而只是指代程序中某个元素的名称，仅是在定义时才绑定某个具体的值，只是对值的一次引用，以便后续对该值进行各种操作。而且这种微弱的绑定关系随时可以改变。例如：

```
julia> a = 10
10

julia> a = 3.2
3.2

julia> a = "this is not a number"
"this is not a number"
```

其中，变量 `a` 先是用于指代整数 10，再用于指代 3.2 这个浮点数值，最后又变成了字符串。

所以在 Julia 中，变量本身没有类型的概念，值（对象）才有。如果说一个变量是某类型，实际是指其引用的值属于该类型，这与 C++ 或 Java 这些强类型语言是完全不同的。在这些强类型语言中，变量需要以声明的方式与某类型建立确切、唯一且恒定的关系，之后这个变量无法再变更类型，只能存取操作该类型的值。

在 Julia 这种弱类型机制中，变量无须先声明即可随时使用，而且不强制进行类型的限定。在语言内部，编译期的类型检查也不会太严格，一般直到运行时才确定操作目标对象的具体类型。这也是动态语言的基本特点。而且，变量的类型可以像上例那样随时变换（虽然这并不是 Julia 建议的做法）。

然而，过于松散的类型控制（例如允许数据类型随意变换或整型可与字符值相加），很



容易导致开发的程序出现莫名其妙的错误。同时，如果类型延迟到运行期才确定，往往会带来很多效率损耗。所以 Julia 的编译器不会自动地对类型做隐式转换或提升，除非转换提升存在着明确的定义（语言内置的或用户定义的代码中）；而且为了科学计算的高性能，Julia 建议开发者在使用变量或值时，要尽可能地给出确切、详尽的类型描述，并能够维持类型的稳定性。

对于变量，如果希望其与值的绑定关系稳定，可以在定义时使用关键字 `const` 在定义时将其标识为常量。例如：

```
julia> const mye = 2.71828182845904523536;  
julia> const mypi = 3.14159265358979323846;
```

其中，`mye` 和 `mypi` 均被定义为浮点数常量。

原则上，常量只能在声明时赋值一次。但正如上文所言，变量与值之间的赋值实际只是建立了两者的绑定关系，所以 `const` 在本质上影响的便是这种微弱的绑定关系。如果一定要对常量重新赋值，也是可以的，例如：

```
julia> mye = 3.21  
WARNING: redefining constant mye  
3.21
```

```
julia> mye  
3.21
```

其中，强制性地修改了常量 `mye` 的值，但在返回新值前会报出“redefining”警告信息，不过仍会执行成功。但需要注意的是，如果试图将普通的同名变量再次定义为常量，会报异常（而不仅仅是个警告），即会执行失败：

```
julia> mypi = 3.14159  
3.14159  
  
julia> const mypi = 3.1415926  
ERROR: cannot declare mypi constant; it already has a value
```

6.4.2 类型断言

弱类型在对变量控制方面更为简单、轻松，但在计算操作中，仍需要明确的数据或值的内存结构，即表明了数据或值内部表达方式的类型定义。在开发中，必要时可通过操作符 `::` 对值的类型进行“断言”，以判断其是否为某个类型。例如：

```
julia> 1::Int64  
1
```

```
julia> a = Int32(1)  
1
```

```
julia> a::Int32  
1
```

```
julia> a::Int64  
ERROR: TypeError: typeassert: expected Int64, got Int32
```



可见断言表达式会在类型相容（匹配）时返回原值，否则上报 `TypeError` 异常。

这种断言操作对抽象类型同样有效：如果值的具体类型是被断言类型的子类型，断言也会成立。例如：

```
julia> 1::Number
1
```

```
julia> a::Real
1
```

当然，在断言之前我们也可以通过 `isa()` 函数判断某个值是否为给定类型的实例，例如：

```
julia> a = 1;
```

```
julia> isa(1, Int32)
false
```

```
julia> isa(a, Int64)
true
```

这样，在预先知道值不是指定类型的对象时，可提前给予恰当的处置，不会像断言那样让程序报错中断。

同样，该函数也适用于抽象类型，例如：

```
julia> isa(1, Integer)
true
```

```
julia> isa(2.3, Float64)
true
```

```
julia> isa(2.3, Real)
true
```

```
julia> isa(2.3, Any)
true
```

此时若第一个参数（值）的具体类型是第二个参数（类型）的子类型，`isa()` 便会取 `true` 值，否则会返回 `false` 值。

6.4.3 DataType

在 Julia 中一切皆对象，类型本身也是可以操作的对象。如果我们将某类型作为参数，使用 `typeof()` 查看其类型，会发现：

```
julia> typeof(Int64)
DataType
```

```
julia> typeof(Integer)
DataType
```

```
julia> typeof(Any)
DataType
```

其中，`DataType` 便是 Julia 中包括元类型、抽象类型及后文介绍的复合类型等所有类型的



“类型”；而且也是 Any 的类型（换言之，Any 是其实例之一）。事实上，DataType 本身也是类型的一种，在类型系统中也是 Any 的子类型之一。

对于类型对象，前述的 isa() 函数与断言操作符 :: 同样适用，例如：

```
julia> isa(Int, DataType)
true

julia> isa(Any, DataType)
true

julia> isa(DataType, DataType)
true

julia> Int64::DataType
Int64

julia> Real::DataType
Real

julia> DataType::DataType           # DataType是DataType自身的类型
DataType
```

可以说，在 Julia 的类型系统中，任意的类型均是 DataType 的实例（对象）。这样的设计机制使得 Julia 的类型系统构成了所谓“完备的闭集”，在概念与操作上具备充分的统一性。

为此，作为可操作的对象，类型之间也可以使用“是否相等”或“完全相同”运算符，判断它们是否是同样的类型，例如：

```
julia> Real == Number
false

julia> Int32 === Int32
true
```

6.4.4 类型别称

类型本身也可以作为右操作数，赋值给某个变量，例如：

```
julia> MyInt = Integer
Integer

julia> MyInt === Integer
true
```

显然，其中定义的类型变量本身仍是 DataType 的实例，也是可用的类型，即：

```
julia> typeof(MyInt)
DataType
```

这种类型变量有时候会非常有用，例如，在长期迭代或规模稍大的程序中，为了实现前后的兼容性、适应第三方包或方便系统移植，往往需要对类型进行各种变化，这其中最简单的方式便是为类型另起一个名字。假设需要达到如下的效果：

```
# 32位系统
julia> UInt
UInt32
```

```
# 64位系统
julia> UInt
UInt64
```

即无论是怎样的系统，都希望 `UInt` 都是有效的类型，而且能自动适配系统的位数，可写如下的代码实现：

```
if Int === Int64           # Int是Julia预先定义的具体类型，会随系统而变化。
    const UInt = UInt64
else
    const UInt = UInt32
end
```

此后，`UInt` 便会在不同的系统中自动取对应的具体元类型，且与源类型有着一致的操作，从而可轻松地实现代码移植。

对于一些结构复杂、表述较长的类型，可以通过这种方法声明一个类型别称，以方便使用。可以说，实现过程中涉及直接的类型操作时，这样的机制提供了很大的想象空间，能够开发出更为灵活自如的程序。

6.4.5 继承关系

类型之间除了能够进行“是否相等”或“完全相同”判断，也能够进行“小于等于”或“大于等于”这种比较操作。不过意义不再是数值的大小，而是在类型拓扑图中，两者是否处于同一条路径上，即两者是否存在父子继承关系。

在实践中，可以使用 `<:` 作为操作符判断一个类型是否是另外一个的子类型，例如：

```
julia> Unsigned <: Integer
true
```

```
julia> Signed <: Real
true
```

```
julia> AbstractFloat <: Any
true
```

或者：

```
julia> Number <: Integer
false
```

```
julia> Real <: AbstractFloat
false
```

显然：

```
julia> DataType <: Any
true
```

这确实是一件很有意思的事情：如前所述，`Any` 的类型是 `DataType`，而 `DataType` 因为

也是一种类型，所以也是 `Any` 的子类型。

另外，对于定义的类型别称，会自动获得源类型在类型系统中的位置，进而获得源类型的继承关系，例如：

```
julia> MyInt >: Int64
true
```

```
julia> MyInt <: Number
true
```

与操作符 `<:` 相对，也可以使用操作符 `>:` 判断一个类型是否是另外一个的父类型，例如：

```
julia> Unsigned >: Integer
false
```

```
julia> Number >: Integer
true
```

```
julia> Real >: Real
true
```

除了结果相反，用法上与 `<:` 是一致的。

如果参与继承关系断言的是同一个类型，会发现：

```
julia> Integer <: Integer
true
```

```
julia> Integer >: Integer
true
```

```
julia> Any <: Any
true
```

也同样是成立的。所以在 Julia 的类型系统中，类型与其自身是满足继承关系断言的，即与自身有父子关系。

6.5 复合类型

一般而言，元类型表达的是单一、纯粹的数值。但在实际的开发过程中，我们往往需要对各类事物进行建模，用于描述更为复杂多样的数据类型，例如，一个三维的地理坐标，不同制造商、颜色、发动机型号的汽车，等等。

为此，需要有一种结构能够将多个概念组合封装，用于表达多属性的事物模型。复合类型是 Julia 中可具有成员字段的复合结构，为开发者提供了自定义类型的机制，与元类型等基本类型有着相同的独立地位，也是 Julia 面向对象编程的基础，在 Julia 中有着广泛的应用。

6.5.1 基本定义

Julia 中，复合类型均通过 `struct` 关键字^①进行声明，基本语法为：

① Julia 在 v0.6 之前并没有 `struct`，而是使用 `type` 定义复合类型，这是一个很大的变化。

```
[mutable] struct 类型名
    成员字段1::类型1
    成员字段2::类型2
    # 其他成员字段
end
```

其中，语法中的操作符 `::` 会限定成员字段为指定类型，不过是可以省略的。当省略时，成员类型默认为 `Any` 类型，意味着该字段变量能够绑定到任何已有的类型，包括元类型、抽象类型甚至是复合类型本身。关键字 `mutable` 是可选的标识符，不用时意味着该 `struct` 一旦实例化，成员取值将不能再进行任何变更；使用时则说明该 `struct` 可变，即创建后对象的成员可以反复被修改更新。



提示 如果了解其他面向对象语言会发现，`struct` 内部并没有提供成员函数的定义语法，这是 Julia 有别于其他语言的特色之一：类型仅仅是类型。关键字 `struct` 仅需纯粹地声明复合类型，相关的各种操作方法可在结构外部进行定义，这也是 Julia 设计者建议的规范。

下面我们分别定义不可变和可变两个复合类型，如下：

```
struct FooA                                # 不可变类型
    a
    b::Float64
end

mutable struct FooB                        # 可变类型
    a
    b::Float64
end
```

两者有着相同的字段，`a` 无类型限制，`b` 限定为 `Float64` 类型。

当然，成员也可以是复合类型，例如：

```
struct Bar
    m::FooB
    n::Int64
end
```

对于声明过的类型，我们可以通过内置的 `dump()` 函数查看其内部结构，如下：

```
FooA <: Any
a::Any
b::Float64
```

可见，`a` 被默认为 `Any` 型，同时声明的复合类型的父类型也是 `Any` 型，因为复合类型也是 Julia 中的一种类型。另外一个复合类型 `FooB` 因类同不再列出。

不过，`Bar` 的内部有些特别，如下：

```
Bar <: Any
m::FooB
n::Int64
```

其中，成员 `m` 是声明过的复合类型 `FooB`。

实际上, 如果查看声明的复合类型 FooA 本身的类型, 会发现:

```
julia> typeof(FooA)
DataType
```

也是 DataType 的一种。事实上, 以 struct 声明的复合类型均是 DataType 的实例对象。

对于声明的任意复合类型, 除了使用 dump() 函数外, 我们也可以通过一个专门函数 fieldnames() 获知其有哪些字段, 例如:

```
julia> fieldnames(FooB)
(:a, :b)
```

该函数会在元组中以 Symbol 类型列出所有成员的名称。

6.5.2 默认构造函数

在声明复合类型后, 对其实例化的最简单做法是采用如下的形式:

类型名(成员值1, 成员值2, ...)

其中三点省略符, 指余下还有其他的成员实参值

这与函数(关于函数的更多概念, 参见第7章)的常规调用方式极为相似, 只不过函数名正好是类型名。调用时, 只需在参数列表中按字段定义的顺序逐一给出每个字段的值即可。例如:

```
julia> x = FooA(1, 2.5)
FooA(1, 2.5)

julia> isa(x, FooA)
true
```

便可得到 FooA 的复合类型对象 x 变量, 其中 a 取 1 而 b 取 2.5。

事实上, 在复合类型声明的同时, Julia 内部会自动为其提供默认的构造方法(Constructors), 而且一般会有两种形式。以 FooA 为例, 默认构造方法为:

```
FooA(a, b::Float64)
FooA(a, b)
```

可见构造方法是与类型同名的函数, 而且默认构造方法的参数个数与字段数一致。


这两个默认构造方法的差异为:

- ❑ 前者参数表依据字段定义, 给出了严格的类型限定, 调用者必须提供满足条件的实参值。
- ❑ 后者参数表均是 Any 类型, 接收参数时会自动将其转换到各字段要求的类型, 如果转换失败则报错, 实例化也会失败。

构造对象时, Julia 会依照多态分发原则(后面介绍)在多个构造方法中自动选择。如果构造时提供的参数不满足所有方法的要求, 便会报 MethodError 异常, 例如:

```
julia> FooB()
ERROR: MethodError: no method matching FooB()
Closest candidates are:
  FooB(::Any, ::Float64) at REPL[2]:2      # 默认构造方法(有类型限定)
  FooB(::Any, ::Any) at REPL[2]:2         # 默认构造方法(无类型限定)
```


因为该例在对 `FooB` 实例化时, 参数个数不符合任一构造方法的原型。

 **提示** 出现异常时, Julia 会在错误信息中的 “Closest candidates are” 之后给出可选的方法, 开发者可以依此调整代码, 或者按需自定义构造方法 (后面会介绍)

6.5.3 成员访问及不可变性

构造出复合类型的实例对象后, 便可通过成员访问符 (英文句号) 访问其内部成员, 形式为:

对象名. 字段名

 **注意** “对象名” 与 “成员访问符” 之间不能有空格。

对于上文中 `FooA` 对象的 `x` 变量, 有:

```
julia> x.a
1
```

```
julia> x.b
2.5
```

但是如果试图修改其成员的内容, 会报错:

```
julia> x.a = 10
ERROR: type FooA is immutable
```

```
julia> x.b = 3.8
ERROR: type FooA is immutable
```

这是因为 `FooA` 在声明时, 没有使用 `mutable` 进行标识。不过, 前文中的 `FooB` 类型在声明时, 是有 `mutable` 标识的。我们创建一个 `FooB` 对象, 如下:

```
julia> y = FooB(2, 3.5)
FooB(2, 3.5)
```

```
julia> y.a
2
```

```
julia> y.b
3.5
```

再尝试修改其成员值:

```
julia> y.a = 2.9                                     # 类型由Int64变为Float64
2.9
```

```
julia> y.b = 2//3
2//3
```

```
julia> y
FooB(2.9, 0.6666666666666666)                       # 有理数被提升为浮点数
```

可见, 成员被成功修改了。

此外，由于 `y` 中的字段 `b` 限定了类型，所以即使修改时提供了别的类型，新的值也会保持类型不变。但这种操作需要保证提供的新值能被提升为限定的类型，否则会报错，例如：

```
julia> y.a = 3.2          # 字段a是Any类型，所以正常执行
3.2

julia> y.b = 3+2im        # 字段b限定为浮点型，但给定的复数型是比浮点型“更大”的类型，故错误
ERROR: InexactError: Float64(Float64, 3 + 2im)
```

但是，可以将成员的类型限定为抽象类型，以提高其相容性，例如：

```
julia> mutable struct FooC
    a::Int32
    b::Real          # 限定为抽象类型，可以为实数的任意子类型
end

julia> w = FooC(1, 2);

julia> typeof(w.b)
Int64              # 实际类型是Int64

julia> w.b = 3.2;

julia> typeof(w.b)
Float64           # 类型被改变
```

其中，定义的 `FooC` 成员 `b` 限定为 `Real` 类型，所以赋值为 `Int64` 或 `Float64` 类型均可。显然，成员 `b` 仍不能赋值为复数型，即：

```
julia> w.b = 3+2im        # Complex{Int64}
ERROR: InexactError: Real{Real, 3 + 2im}
```

会报错，因为复数不是 `Real` 类型的子类型。

如果成员有复合类型的 `Bar`，将上例中的 `FooB` 对象 `y` 作为参数对其实例化，例如：

```
julia> z = Bar(y, 20)
Bar(FooB(2.9, 0.6666666666666666), 20)
```

详细查看其成员内容，如下：

```
julia> z.m
FooB(2.9, 0.6666666666666666)

julia> z.n
20
```

同样，`Bar` 对象 `z` 也是不可变的：

```
julia> z.m = FooB(2.3, 3.8)
ERROR: type Bar is immutable

julia> z.n = 30
ERROR: type Bar is immutable
```

但其内部的 `FooB` 却是可变的，即：

```
julia> z.m.a = 2.5
2.5
```

```
julia> z.m.b = 5.0
5.0
```

可见, `struct` 结构的标识符 `mutable` 只控制着本层的不可变性, 不会波及内层或外层, 即不可变性不会在上下层之间传播。

不可变的复合类型是 Julia 对结构内权限进行控制的一种方式。相对于可变复合类型, 编译器在处理不可变类型时能够高效地处理内部的存储结构, 而且能够推导出代码中使用了哪些不可变对象, 从而提高了编译效率。

另外, 正因为不可变对象内部成员的值是恒定的, 所以值的组合便可用于区分不同的对象。相对地, 可变对象的内容随时都会改变, 不变的仅是其在堆中的内存地址, 所以地址是区分它们的唯一可信的依据。更需要注意的一点是, 不可变对象在赋值或函数传参时均采用“值拷贝”的方式, 而可变对象则会采用引用的方式。



提示 在开发过程中, 使用可变对象还是不可变对象, 可以参考以下两点:

- ☐ 当两个对象具备相同的成员值集合时, 是否需要识别为不同的事物。
- ☐ 对象之间是否会随时独立地进行值的变换。

如果以上的答案都是“否”, 则建议使用不可变类型。

6.5.4 单例复合类型

没有任何成员字段的不可变复合类型会成为单例 (Singleton), 即以其为基础创建的任意对象实际都是一样的, 没有区别。换言之, 单例类型有且仅有一个实例对象。例如:

```
julia> struct NoFields1                                # 声明空类型 NoFields1
end
```

```
julia> struct NoFields2                                # 声明空类型 NoFields2
end
```

对创建 `NoFields1` 的两个对象进行对比:

```
julia> NoFields1() === NoFields1()
true
```

可见两者是“完全相同”的。显然, 对于 `NoFields2` 类型也同样如此。

但是, 不同名的 `struct` 单例是不同的对象, 即:

```
julia> NoFields1() === NoFields2()                    # 不满足完全不同的要求
false
```

```
julia> NoFields1() == NoFields2()                      # 内容虽然都是空的, 但却不同
false
```

其实这点不难理解, 类型名不同所以对象不同; 而类型名相同时, 既然两个对象都是零字段, 自然在对比时内容也是相同的。

但需注意的是，被 `mutable` 关键字修饰的空字段的可变复合类型是不会成为单例类型的，例如：

```
julia> mutable struct NoMember1 end

julia> NoMember1() === NoMember1()
false

julia> NoMember1() == NoMember1()
false
```

虽然 `NoMember1` 的两个对象类型一致而且都是零字段，但却不是相同的对象，甚至都不相等，这点需要特别注意。

6.6 类型联合

有时候，无论是类型的别称还是抽象都难以满足程序设计的需求。例如某个函数的参数，我们希望能是 `Int64` 或 `Float32` 型，但不能是任何其他类型。这样的要求，采用前文介绍的方式是难以实现的。

Julia 中提供了一种可以将多种类型进行“或组合”的方式，即类型联合 (Type Unions)，定义的基本语法为：

```
Union{T1, T2, ...}
```

其中，`Union` 会将大括号内以逗号分隔的多个类型进行组合，变成单一的类型。

例如，我们可以声明一个内部类型元素为 `Int64` 与 `Float32` 的类型联合，代码如下：

```
julia> IntOrFloat64 = Union{Int64, Float64}
Union{Float64, Int64}

julia> typeof(IntOrFloat64)
Union
```

此时对其进行相关类型的继承关系断言，便会发现除了 `Int64` 与 `Float64` 外，其他的类型都不是其子类型：

```
julia> Int64 <: IntOrFloat64
true

julia> Float64 <: IntOrFloat64
true

julia> FooA <: IntOrFloat64      # FooA是前文曾声明过的复合类型
false

julia> Real <: IntOrFloat64
false
```

如此一来，便实现了对多个类型进行组合限定的目的。

在类型联合的基础上，我们便可用其限定具体值的类型，只有满足类型断言的对象才

能够获得有效操作。例如：

```
julia> 1::IntOrFloat64
1
```

```
julia> 2.3::IntOrFloat64
2.3
```

```
julia> a = Int32(1)
1
```

```
julia> a::IntOrFloat64
ERROR: TypeError: typeassert: expected Union{Float64, Int64}, got Int32
```

其中，变量 `a` 因为类型是 `Int32`，既不是 `Int64` 也不是 `Float64`，所以会上报 `Type Error` 异常。

`Union` 在元素类型上并没有过多限制，不仅适用于基本的类型，也适用于其他更多的类型。例如：

```
julia> MockType = Union{FooA, FooB, Integer};
```

其中，`MockType` 将复合类型与普通的抽象类型进行了联合，构造出了新的类型。将其与其他各种类型对比断言，便会发现其确实对元素中的类型有了准确的限定约束，代码如下所示：

```
julia> FooA <: MockType
true
```

```
julia> FooB <: MockType
true
```

```
julia> MockType >: Signed           # 其中的Integer是Signed的父类型
true
```

```
julia> MockType >: AbstractFloat
false
```

```
julia> MockType >: Bar               # Bar是前文曾声明过的复合类型
false
```

另外一个事实是，虽然类型联合的类型是 `Union` 而非是 `DataType`，但 `Union` 的类型却是 `DataType`，即：

```
julia> typeof(Union)
DataType
```

所以类型联合也是 `DataType` 的实例对象，同样也是类型系统的一部分。

但是，`Union` 的直接父类型并非 `Any` 类型，而是：

```
julia> supertype(Union)
Type{T}
```

其中，`Type{T}` 是一种特殊的参数化类型，将会在接下来的内容中进行介绍。

6.7 TypeVar

1. 最底层类型

如果 `Union` 内部无类型元素，即参数为空，`Union{}` 便成了 Julia 中的另外一种特殊用法，用于表示类型系统中“最底层”的类型，即 `Union{}` 是所有类型的子类型，位于拓扑树的底部。例如：

```
julia> Union{} <: Int8
true
```

```
julia> Union{} <: Real
true
```

```
julia> Union{} <: Char
true
```

```
julia> Union{} <: Any
true
```

```
julia> Union{} <: Type
true
```

所以，Julia 中的任意类型 `T` 均满足：`Union{} <: T <: Any`。如此一来，Julia 的类型系统就构成了包含所有类型为元素的闭集。

需要注意的是，Julia 中的 `Union{}` 是独立的类型，且其类型为 `Core.TypeofBottom`，而不是 `DataType`，即：

```
julia> typeof(Union{})
Core.TypeofBottom
```

不过 `Core.TypeofBottom` 的类型仍是 `DataType`，即：

```
julia> typeof(Core.TypeofBottom)
DataType
```

2. 类型范围

如上所述，在 Julia 的类型系统中，`Any` 是所有类型的父类型，位于类型拓扑树的顶部；而 `Union{}` 则是所有类型的子类型，位于拓扑树的底部。

前文曾提及，在类型拓扑图中，从任一类型节点到 `Any` 类型之间只存在唯一的路径，而且路径上的任意两个类型之间都存在父子关系。通过 `Union` 虽然可以取得一组类型组合，但需要逐一罗列才行。

不过，如果我们不需要跨路径的类型进行组合，而只需要同路径的诸多类型，是完全可以不必罗列的。Julia 内部提供了一个 `TypeVar` 类型，用于表达某类型路径上一段范围的所有可选类型。其内部结构为：

```
TypeVar <: Any
  name::Symbol          # Symbol 类型后文介绍
  lb::Any
  ub::Any
```

可见 `TypeVar` 有三个字段：其中 `name` 是类型变量的名称；成员 `lb` 限定了 `name` 的类型下界，即 `name` 必须是 `lb` 的父类型；`ub` 则指定类型上界，即 `name` 必须是 `ub` 的子类型。

事实上，作为 `DataType` 实例的类型本身，`TypeVar` 提供了一种对这些实例进行遍历的复合结构。其构造方法原型为：

```
TypeVar(name::Symbol, ub::ANY, lb::ANY) # 注意上界在下界之前，其中的::用于限定参数类型（后文介绍）
```

如果在构造时，其中的参数 `lb` 与 `ub` 省略，则表示不限定类型的上下界，此时这两个参数分别默认为 `Any` 和 `Union{}`，即：

```
julia> Union{} <: TypeVar{::T} <: Any
true
```

如果要表达类型拓扑图中某段确切的类型范围，则可指定具体的类型作为上下界。例如：

```
julia> T = TypeVar{::T, Signed, Real}
Signed<:T<:Real
```

表示类型 `T` 可以是 `Signed` 到 `Real` 中的任一类型，此时其内部结构为：

```
TypeVar
  name: Symbol T
  lb: Signed <: Integer
  ub: Real <: Number
```

一般来说，`TypeVar` 会被 Julia 隐式地使用，开发者不需要显式地使用该类型。在我们对类型进行参数化时，这种结构便会由内部自动创建。下面会详细介绍类型参数化方法。

6.8 类型参数化

对于有成员结构的类型，往往需要限定内部成员的类型。但很多时候，在结构内部进行类型限定时会使复合类型的适用性受到限制。为了不放弃类型的限定条件，又适应灵活的类型使用场景，便有了对类型进行参数化的需求。

例如，需要定义一个坐标点类型，会有一维、二维、三维等情况；而每一维的类型可以是 `Int8`、`Int64`，也可以是 `Float32`、`Float64` 等。如果有严格的类型限定，需要针对不同的类组合定义很多具体的复合类型，而且一旦需要为这些类型定义操作函数，也会需要众多的实现方法。这种做法不但导致代码量骤增，维护起来也很麻烦。

为此，我们需要对概念进行抽象。例如，可定义一个无维度限定的名为 `Pointy` 的抽象类型，为所有的点类型定义统一的接口函数；然后再定义一维点 `Point1D`、二维点 `Point2D` 及三维点 `Point3D` 等复合类型，针对不同维度定义相应的操作方法。在确定成员类型时，将成员的类型作为参数，交由外部调用方控制。

6.8.1 参数化复合类型

1. 定义

若要对复合类型实现参数化，只需声明时在类型名称之后附加类型参数（下文简称类

参) 列表即可, 如下所示:

```
[mutable] struct 类型名{T1, T2, ...}
    成员字段1::T1
    成员字段2::T2
    # 其他成员定义
end
```

其中, mutable 为可选; 类参需在紧跟类型名的大括号 { } 内列出 (类型名后不能有空格); 内部的成员字段若要限定类型, 便可以在类参表中选择一个恰当的类型使用。当然, 也可以不借助类参, 仍可以使用某个明确的类型对其单独地限定, 而不使用类参, 或者干脆不限定类型。

在对参数化的复合类型实例化时, 类型实参需是已经存在的类型, 可以是元类型、抽象类型, 甚或某种复合类型等。

让我们继续前述坐标点的例子。先尝试对一维点类型 Point1D 实现参数化, 如下:

```
julia> mutable struct Point1D{T}
    x::T          # 限定成员类型为T
end
```

此后, 通过给定具体的类参 T, 便可基于上面的参数化类型定义出具体的复合类型 (为方便, 称之为具象化类型), 例如:

```
julia> Point1D{Int32}
Point1D{Int32}

julia> Point1D{Real}
Point1D{Real}
```

其中, 打印的类型信息中多出了 {Int32} 或 {Real} 标识, 表达了参数化类型被具象化之后的实际类型, 遵循着类参提供时的语法格式。在上例中, 类参 T 被具体的实参 Int32 或 Real 类型替代, 生成了两种具象化类型。

对于得到的每个具象化类型, 实际上“复制”了其参数化版本的结构, 并同时将其中的类参落实为某个确切的类型, 从而得到一个新的类型。如对于上例中的 Point1D{Int32}, 实际上得到了内部结构如下的复合类型:

```
Point1D{Int32} <: Any
x::Int32
```

因为 x 在参数化类型中被限定为类参 T, 所以在得到的具象化类型中, x 的类型被实际限定为了 Int32 类型。

从某个角度来说, 因为类参的扩展性及具象时的多样性, 声明的参数化类型其实相当于一个名称相同的类型族。而且, 在类型关系上, 定义出的这些具象化类型均是参数化类型的子类型, 即:

```
julia> Point1D{Int32} <: Point1D
true

julia> Point1D{Real} <: Point1D
true
```

不过，“族”中的各具象化类型之间并不存在父子关系，即使它们的类参之间存在父子关系。例如：

```
julia> Point1D{Int32} <: Point1D{Real}
false

julia> Point1D{Real} <: Point1D{Int32}
false
```

虽然其中的 `Real` 是 `Int32` 的父类型，但对应的具象化类型不会自然地延续这种关系。显而易见，各具象化类型也不满足“相等”或“相同”的判断，例如：

```
julia> Point1D{Int32} == Point1D{Real}
false
```

即类型参数不同，对应的具象类型便不同。

参数化类型与具象化类型的关系，可以类比于 `DataType` 与 `Int32`、`Real` 等类型之间的关系。每个具象化类型都是参数化类型的实例，而且也都是独立的类型。

2. 实例化

事实上，不论是 `Point1D` 还是某个其他的具象化类型，我们都能够发现 Julia 内部为其提供的默认构造方法中形如以下类型：

```
(::Type{Point1D})(x::T) where T
```

看起来与前述那种无类参的复合类型有不少差异：一个是类型名被 `::Type{}` 结构所封装，另一个是其中出现了 `where T` 结构。这样的结构我们会在后文详细介绍，姑且暂时忽略它，便可发现这仍是一种函数的形式，所以依然能够以类似普通复合类型的方式对其进行实例化，例如：

```
julia> p1 = Point1D(1.1)
Point1D{Float64}(1.1)
```

不难理解，输入的参数 `1.1` 用于给成员 `x` 赋值，因为其被自动识别为 `Float64` 类型，所以上例中这种没有明确给定类参的创建方法，会自动将 `T` 具象化为 `Float64` 类型。若查看上例中成员 `x` 的值与类型，可发现：

```
julia> p1.x
1.1

julia> typeof(p1.x)
Float64
```

其中，`x` 确实为提供的值，而其类型也与提供值的类型一致。

如果我们变换提供值的类型，那么信息也会随之而变，例如：

```
julia> p2 = Point1D{Int32}(1)           # 限定成员参数为Int32类型
Point1D{Int32}(1)

julia> p2.x
1
```



```
julia> typeof((p2.x))
Int32
```

可见在对参数化类型实例化时，可不必先对其具象化，直接提供成员值便可创建出具体的对象，而类参的实际具象类型会自动推断出来。

当然，实例化时类参的实际类型也可以不由 Julia 自动推断生成，而是显式地给出：

```
julia> p2 = Point1D{Int32}(Int64(1))    # 明确地限定提供值1为Int64类型，但类参具象为Int32
Point1D{Int32}(1)                      # 对象的实际类型取Int32
```

```
julia> typeof(p2.x)
Int32
```

此时成员参数的类型优先级较低，最终的类型会由类参的具象类型决定。

在此，针对参数化类型的实例化方法，做以下总结：

```
类型名(成员值1, 成员值2, ...)          # 采用同非参数化的普通类型一致的实例化方法
类型名{T1, T2, ...}(成员值1, 成员值2, ...) # 明确地列出类型实参
```

以上两种方式都可以对参数化的类型进行实例化。

下面我们再看具象化的另外一个特性：类似于非参数化复合类型，具象化类型的成员类型是稳定的，例如：

```
julia> p2.x = Int8(2)                  # 赋值为Int8类型
2

julia> typeof(p2.x)
Int32                                  # 保持不变

julia> p2.x = 1.2                      # 赋值为Float64类型
ERROR: InexactError: Int32{Int32, 1.2}
```

例中试图在赋值时改变 $x::T$ 的类型，不过，都在可提升的情况下被隐式地转为类参限定的类型。但如果成员参数值的类型无法转换或提升为具象类型，这种改变类型的赋值操作不会成功而且会报异常。所以，在实例化时，如果采用前面第二种方法，作为整个复合类型的类参，会在类型限定方面有着更强的约束，类参限定的成员是无法改变类型的。

下面我们再看三维点的例子，其声明方法如下：

```
julia> mutable struct Point3D{T1, T2}
    x::T1
    y::T1
    z::T2
end
```

其中有 $T1$ 与 $T2$ 两个类参，成员 x 与 y 均被限定为类型 $T1$ ，而 z 则限定为另一类型 $T2$ 。这意味着，成员 x 与 y 必须是相同的具体类型，而 z 可以是与之不同的其他类型；当然，在具象化时 $T1$ 与 $T2$ 一致，此时三个成员的类型便是一致的。

此时 `Point3D` 的默认构造方法形式如下：

```
(::Type{Point3D})(x::T1, y::T1, z::T2) where {T1, T2}
```

可见在参数表中， x 与 y 的类型是相同的。若对其实例化，可得：

```
julia> q1 = Point3D{Int64, Float32}(1, 2, 3.1)    # T1与T2分别被具象化为Int64和Float32
Point3D{Int64,Float32}(1, 2, 3.1f0)

julia> typeof(q1.x) == typeof(q1.y) == Int64
true

julia> typeof(q1.z)
Float32                                     # 3.1被赋值为z后, 被转为Float32类型
```

其中, x 与 y 的类型均是 $T1$ 的实参(具象类型) Int64 , 而 z 的类型则是 $T2$ 的实参 Float32 类型。如果提供相同的 $T1$ 与 $T2$, 则三者会是同样的类型, 例如:

```
julia> q2 = Point3D{Int32, Int32}(1,2,3)
Point3D{Int32,Int32}(1, 2, 3)

julia> typeof(q2.x) == typeof(q2.y) == Int32 == typeof(q2.z)
true
```

可见三者虽然以不同的类参限定, 但当参类相同时, 实例化之后获得了相同的类型限定。

但提供的前两个成员值类型不同时, 对象会创建失败, 例如:

```
julia> Point3D{Int32(1), Int8(2), Float32(3.1)}
ERROR: MethodError: no method matching Point3D{::Int32, ::Int8, ::Float32}
```

虽然前两者都是整型的一种, 而且 Int8 能被提升为 Int32 , 但因类型限定关系是通过 `struct` 传递的, 所以不会成功。

事实上, 有理数型与复数型都是参数化的复合类型, 只不过一个是 Real 的子类型, 另一个是 Number 的子类型, 即:

```
Rational{T<: Integer} <: Real
Complex{T<: Real} <: Number
```

如果查看它们的内部结构, 便可发现:

```
Rational{T<:Integer} <: Real
  num::T
  den::T

Complex{T<:Real} <: Number
  re::T
  im::T
```

它们都有两个成员字段, 但类参却都只有一个, 从而能够确保内部的两个成员必须是一致的类型。

另外, 这两个类型虽然是复合类型, 但都是 Number 的子类型, 即都是数字。它们不但在数学上有着切实的意义, 也确实“共享”着各种数字集合的计算规则。在 Julia 中, 复合类型与普通类型相比, 并不是非常“特别”的结构, 在语法操作上完全可以同等对待。

至此, 我们总结参数化复合类型的特点如下: 复合类型的类参数量不限, 视需要而定; 内部成员可沿袭其类参作为限定类型; 通过类参的组合, 在成员可取任意类型的情况下, 能够限定某些成员必须类型一致。



6.8.2 参数化抽象类型

1. 定义

在 Julia 中，若要定义一个参数化的抽象类型，基本语法为：

```
abstract type 类型名{T1, T2, ...} end
```

其中，“类型名”是待定义的抽象类型名称，大括号中的是各种类参列表。



注意 语法中的“类型名”与左大括号之间不能有空格。

仍继续关注坐标点的例子。假设为二维点定义 Point2D 类型如下：

```
julia> mutable struct Point2D{T}
    x::T
    y::T
end
```

之后，对其定义求向量模的计算：

```
mymodule(p::Point2D) = sqrt(p.x^2 + p.y^2)    # 定义了名称为mymodule的函数（后文介绍）
```

显然调用时需提供 Point2D 类型的参数才能正常获得结果，例如：

```
julia> p1 = Point2D(1, 2);
```

```
julia> mymodule(p1)
2.23606797749979
```

而采用 Point1D 等其他类型是不允许的，例如：

```
julia> mymodule(Point1D(2.2))
ERROR: MethodError: no method matching mymodule(::Point1D{Float64})
```

在以 Point1D 类型作为参数调用时会报 MethodError 错误，并提示未找到匹配的方法。可以想见，对 Point3D 也同样如此。

当然此函数的参数 p 也可以限定为 Point2D 的某个具象类型，例如：

```
mymodule(p::Point2D{Int64}) = sqrt(p.x^2 + p.y^2)
```

显然此方法限制性过大，徒增烦恼；而之前以参数化类型的名称限定 p 的类型更具普适性。

如果我们需要对前文中的 Point1D 与 Point3D 也提供同样的求向量模功能函数，显然需为这两个类型提供单独的定义，即再定义两个计算函数。如果应用中存在更多维的坐标类型，同时功能函数不仅仅求模值这一种，那么代码量将骤增，而且是枯燥地重复性增加，会给后续的维护带来麻烦。所以，我们需要一种抽象的坐标类型，能够承载任意维度坐标共有的计算操作。

为此定义一个名为 Pointy 的抽象类型，并进行参数化，如下：

```
julia> abstract type Pointy{T} end
```

如果 T 被指定，获得的所有具象化类型均为 Pointy 的子类型，例如：

```
julia> Pointy{Int64} <: Pointy
```



```
true
```

```
julia> Pointy{Float64} <: Pointy
true
```

更为灵活的是，参数 T 还可以取具体的数值：

```
julia> Pointy{1} <: Pointy
true
```

这对于内部成员有元素数量可动态延展的结构（如后文介绍的数组），会非常有用。

对于声明的参数化抽象类型，不同具象类型之间同样无父子关系，例如：

```
julia> Pointy{Float64} <: Pointy{Real}
false
```

虽然 Real 是 Float64 的父类型，但以它们为基础的具象类型不具有相容性。

2. 应用

实际上，如抽象类型或元类型那样，类型之间的继承关系是需要在声明时显式确定的。对于之前声明过的 Point1D 、 Point2D 与 Point3D ，完全可以在声明时作为抽象坐标类型 Pointy 的子类型。所以将它们的声明方式修改如下：

```
julia> mutable struct Point1D{T} <: Pointy{T}
    x::T
end
```

```
julia> mutable struct Point2D{T} <: Pointy{T}
    x::T
    y::T
end
```

```
julia> mutable struct Point3D{T} <: Pointy{T}
    x::T
    y::T
    z::T
end
```

此时三种的参数化类型便成为父参数化类型 Pointy 的子类型，即：

```
Point1D <: Pointy
Point2D <: Pointy
Point3D <: Pointy
```

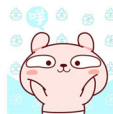
又因为具象类型是参数化类型的子类型，所以子参数化类型的具象类型均是父参数化类型的子类型，即如下的父子关系断言也是成立的：

```
Point1D{Float64} <: Pointy
Point2D{Integer} <: Pointy
```

但父子参数化类型的具象化类型之间仍不存在父子关系，例如：

```
julia> Point2D{Integer} <: Pointy{Real}
false
```

虽然其中的类参具备父子关系。



如此一来，我们便可以在 `Pointy` 上定义一个统一的操作函数，便能适用于三个不同维度的坐标点类型，而不需要更多针对性的重复性函数。重新定义向量求模函数如下：

```
function module_t(p::Pointy) # 函数的定义方式会在后文详述
    m = 0
    for field in fieldnames(p) # 遍历成员字段名
        m += getfield(p, field)^2 # 取得该成员值并平方
    end
    sqrt(m) # 将多成员值平方和开方
end
```

该函数能够适应不同数量字段的复合类型。此时便可用其求解不同维度坐标点的模值，例如：

```
julia> module_t(Point1D{Int64}(1))
1.0

julia> module_t(Point2D{Float64}(2, 2))
2.8284271247461903

julia> module_t(Point3D{Complex}(1+2im, 3+4im, 5-6im))
2.9389894877329246 - 5.44404805351722im
```

可见，子类型沿袭父类型的计算规则能够给开发带来巨大的便利。

不过上例因涉及反射 (Reflection) 机制，未必是最佳实践，仅作为示例使用。

3. `Type{T}`

前面提及，参数化类型 `Type{T}` 是 `Union` 及 `Core.TypeofBottom` 的父类型。实际上，该类型同时也是 `DataType` 的父类型，即：

```
julia> supertype(DataType)
Type{T}

julia> supertype(Type)
Any
```

而 `Type{T}` 的父类型才是 `Any` 类型。

在 Julia 内部，`Type{T}` 是一种非常特殊的参数化抽象类型，也是单例类型的一种，可以认为是类型的生成器。从定义层面讲，每个具象的 `T`，都是 `Type{T}` 类型的唯一实例对象。为便于理解，先看一些例子：

```
julia> isa(Float64, Type{Float64})
true

julia> isa(Real, Type{Real})
true

julia> isa(Point2D, Type{Point2D})
true

julia> isa(Pointy, Type{Pointy})
true
```





但对于不一致的 T ，这种判断不会成立。例如：

```
julia> isa(Float64, Type{Real})
false

julia> isa(Point2D, Type{Pointy})
false
```

而且具体的值也不是 Type 的实例：

```
julia> isa(1, Type)
false

julia> isa("foo", Type)
false
```

可以说，若 $\text{isa}(A, \text{Type}\{B\})$ (A 是 $\text{Type}\{B\}$ 的实例) 成立，有且仅有一种情况： A 与 B 完全相同。

如果结构中没有参数 T ， Type 便是一个简单的参数抽象类型；此时，所有的类型都将是它的实例对象，也包括 Type 本身，即：

```
julia> isa(Type{Float64}, Type)
true

julia> isa(Float64, Type)
true

julia> isa(Real, Type)
true

julia> isa(DataType, Type)
true

julia> isa(Type, Type)
true
```

在下文说明参数化原理时，我们还会介绍另一种 $\text{Type}\{T\}$ 单例类型 UnionAll 。

6.8.3 参数化元类型

元类型同样可以进行参数化，声明的基本语法为：

```
primitive type 类型名{T} bits end
```

其中，`primitive type` 为声明关键字， T 是类型参数。

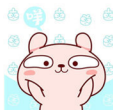
同样，具象化的元类型均是参数化元类型的子类型；不同的具象元类型无父子关系，而且是独立的、不同的类型。

例如，Julia 中有个元类型 `Ptr`，其定义为：

```
primitive type Ptr{T} 32 end      # 32位系统
primitive type Ptr{T} 64 end     # 64位系统
```

当 T 不同时，便是不同的 `Ptr` 类型。

该类型专门用于在混合编程时兼容 C/C++ 语言中的指针类型。实际应用中，可以把





`Ptr` 理解为 C/C++ 语言中星号的作用, `T` 不同便声明了不同指针类型, 意味着 `Ptr` 所指内容的存储结构是不同的。例如 `Ptr{Int64}` 对应于 C/C++ 中的 `int*` 用法, 表示 `int` 型的指针; 而 `Ptr{Float64}` 相当于 `double*` 型指针。

因为参数化元类型在特性上与复合类型、抽象类型一致, 所以此处不再赘述。

6.8.4 参数化基本原理

如果我们查看声明的参数化抽象类型 `Pointy` 的类型, 会发现:

```
julia> typeof(Pointy)
UnionAll

julia> typeof(UnionAll)
DataType

julia> supertype(UnionAll)
Type{T}
```

其中, `Pointy` 的类型不是 `DataType` 而是 `UnionAll`; 而 `UnionAll` 的类型才是 `DataType`; `UnionAll` 的父类型正是前文曾介绍过的 `Type{T}` 类型。

至此, 我们已经遇到了 `Type{T}` 的四个主要子类型: `DataType`、`Union`、`UnionAll` 以及 `Core.TypeofBottom` 类型。它们之间的 `typeof` 与 `supertype` 关系如图 6-3 所示。图中实线箭头由父类型指向子类型; 虚线由实例指向其类型。

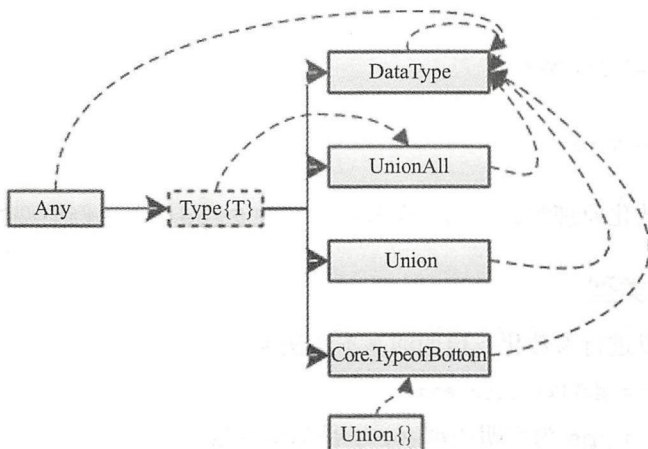


图 6-3 `Type{T}` 类型关系图

由图可见, 参数化的 `Type{T}` 的类型是 `UnionAll`, `Union{}` 的类型是 `Core.TypeofBottom`; 而余下的类型都是 `DataType`, 包括 `DataType` 自身。

事实上, Julia 中的参数化类型都被处理为 `UnionAll` 的实例对象, 其内部结构为:

```
julia> dump(UnionAll)
UnionAll <: Type{T}
```





```
var::TypeVar
body::Any
```

其内部有两个部分：一部分是类型为 `TypeVar` 的 `var`，另一部分是 `Any` 类型的 `body`。

为了方便说明，以其实例 `Pointy` 的内部结构为例，解释这种结构的含义，代码如下：

```
julia> dump(Pointy)
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Core.TypeofBottom Union{}
    ub: Any
    body: Pointy{T} <: Any
```

其中，`body` 实际是 `Pointy` 的原型，而且 `TypeVar` 已被初始化，其 `name` 是声明时的类型参数 `T`，但是其中的类型上下界均采用了默认值，即 `Any` 和 `Union{}` 类型。

如上文所述，`TypeVar` 是可以设定上下界的。如果我们“粗暴”地直接修改 `Pointy` 的上界，然后再次查看其内部结构，会发现：

```
julia> Pointy.var.ub = Real;

julia> dump(Pointy)
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Core.TypeofBottom Union{}
    ub: Real <: Number
    body: Pointy{T<:Real} <: Any
```

其中，`var.ub` 变成了 `Real <: Number`，同时 `body` 内 `Pointy` 的原型变成了：

```
Pointy{T<:Real} <: Any
```

如果我们继续修改下界类型值为 `Integer`，会发现 `Pointy` 的内部结构变成了：

```
julia> dump(Pointy)
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Integer <: Real
    ub: Real <: Number
    body: Pointy{Integer<:T<:Real} <: Any
```

在 `var` 中便意味着 `T` 的上下界分别被限定为 `Real` 与 `Integer` 类型。

如果此时再对 `Pointy` 具象化，会发现已不再是任意的类型都能作为其类参，例如：

```
julia> Pointy{Int64}
ERROR: TypeError: Pointy: in T, expected Integer<:T<:Real, got Type{Int64}

julia> Pointy{Complex}
ERROR: TypeError: Pointy: in T, expected Integer<:T<:Real, got Type{Complex}
```

并提示类型参数 `T` 只能是 `Integer` 的父类型，`Real` 的子类型。

实际上，我们在声明 `Pointy` 时可直接将这种上下界限定关系明确地表示出来，即：

```
abstract type Pointy{Integer <: T <: Real} end
```





如果此时查看该新 Pointy 的内部结构，会发现：

```
UnionAll
  var: TypeVar
    name: Symbol T
    lb: Integer <: Real
    ub: Real <: Number
    body: Pointy{Integer<:T<:Real} <: Any
```

与之前粗暴方式修改过的 Pointy 完全一致。

可见 UnionAll 类型通过将声明的类型原型及 TypeVar 作为成员，便能够实现对参数化类型衍生范围的控制；会基于 TypeVar 提供的约束说明，以类参有效变化范围中的各类型为基础将参数类型展开为一族具象类型，赋予了参数类型类似 Union 的功能；而其中元素类型便是具象类型族。一般而言，有多少类参便会生成多少 TypeVar 成员。

在声明参数化类型时，无论是复合类型、抽象类型或元类型，不放任参数 T 自由选择而给予更为确切的范围限制是非常实用的。例如，在实践中坐标点表达的是图像中的像素位置，要求坐标值为整型而且必须是正数；或是地理经纬度，一般取浮点值，一旦出现其他类型值，通常是无效坐标。如果我们在声明类型时便做出限制，就能够尽早发现异常的数据，确保后续处理尽可能少地出现不可预期的结果或错误。

但是，对于类参范围的限定最好不要通过示例这种粗暴方式（上文仅为示例方便），而应通过声明的方式进行。例如：

```
mutable struct Point{T1 <: Integer, T2<: AbstractFloat}
  x::T1
  y::T2
end
```

或

```
primitive type LittleInt{T<:Integer} 8 end
```

6.8.5 参数化继承关系

在 Julia 中，在声明参数化类型时，其父类型无须是参数化类型。典型的情况是复数型 Complex 及有理数型 Rational，它们都是参数化的复合类型，但其父类型分别是抽象类型 Number 与 Real。不过，如果以某个已经定义的参数化类型作为父类型，类参之间便会存在依存关系，而且声明的方式也有别于普通的类型。

首先，将抽象类型参数化的各种基本语法总结如下：

```
abstract type 类型名{T1, T2, ...} end
abstract type 类型名{T1 <: 类参上界1, T2, ...} end
abstract type 类型名{T1 >: 类参下界1, T2, ...} end
abstract type 类型名{类参下界1 <: T1 <: 类参上界1, T2, ...} end
```

为了方便讲述，上面的语法示例仅以首个类参给出边界限定的表达形式，依次是无边界限定、只上界限定、只下界限定与上下界均限定四种。

声明参数化的复合类型时，如果需要继承某个参数化的抽象类型，有两种基本语法。





为了方便讲述，仍以首个类参示范限界的形式，如下：

```
[mutable] struct 类型名{T1, T2, ...} <: 父抽象类型名{T1, T2, ...}
    # 成员定义
end

[mutable] struct 类型名{T1 <: 类参上界, T2, ...} <: 父抽象类型名{T1, T2, ...}
    # 成员定义
end
```

其中，mutable 同上文，用于控制可变性，为可选关键字；类参的配置有如下几个方面的限制：

- 父类型中类参数量需与本身类型一致，不可多不可少。
- 父类型中类参名均需在复合类型的类参列表中存在。
- 父类型类参中不能再有限界语法，仅需罗列类参名。

而且限界语法只能在复合类型自己的类参列表中实现。另外，复合类型的类参可以更多些，不需一定在父类型中存在，但其类参表必须包含父类型的所有类参名。我们举例如下：

```
abstract type PointWith{T1 <: Integer, T2 <: AbstractFloat} end
abstract type PointNone{T} end

mutable struct PlanePointA{T, V} <: PointWith{T, V}
    x::T
    y::T
    v::V
end

mutable struct PlanePointB{T <: Integer, V <: AbstractFloat} <: PointNone{T}
    x::T
    y::T
    v::V
end
```

其中，声明了两个参数化的抽象类型，并用两种方式分别定义了它们的子类型 PlanePointA 和 PlanePointB。如果分别对 PlanePointA 和 PlanePointB 实例化，会发现：

```
julia> PlanePointA(1, 2, 1)
ERROR: TypeError: AbstractPoint: in T2, expected T2<:AbstractFloat, got Type{Int64}

julia> PlanePointB(1, 2, 1)
ERROR: MethodError: no method matching PlanePointB(::Int64, ::Int64, ::Int64)
```

两种对象的成员 v 的参数值均是 Int64 类型，却都因无法满足类型约束创建对象失败。但错误信息是有差异的，这说明约束检查的节点是不同的。

前一种方法中虽然 PlanePointA 没有无限界语法，但其父抽象类型 PointWith 声明时已经作了限界，所以在建立继承关系时，该限界便会传递到新的子类型 PlanePointA 中，也会在实例化时起效。后一种方法中，父类型 PointNone 虽然未有限界，但 PlanePointB 本身对类型进行了限界操作，所以创建的对象亦需满足约束。



之所以错误不同，因为前者在首先检查父类型类参约束时发现提供的值不符合要求，而后者在调用构造方法时不存在匹配实参值组合的对象创建方法。以笔者之见，更推荐前一种用法，一是可以提前发现问题，二是对错误的定位更准确。

不过还有一种情况：前一种用法中，父抽象类型已经限界，复合类型仍在声明时限定了类参的上下界，例如：

```
mutable struct PlanePointC{T <: Signed, V <: Real} <: PointWith{T, V}
    x::T
    y::T
    v::V
end
```

其中类参 T 在复合类型中“变小”了，而类参 v 变得“更大”了。如果对其实例化，会发现：

```
julia> PlanePointC(1, 1, 2)
ERROR: TypeError: PointWith: in T2, expected T2<:AbstractFloat, got Type{Int64}
```

可见，虽然成员 v 的类型被复合类型放大，但父类型的约束仍在起作用。再例如：

```
julia> PlanePointC(UInt32(1), UInt32(1), 2.35)
ERROR: MethodError: no method matching PlanePointC(::UInt32, ::UInt32, ::Float64)
```

其中，提供的 x 与 y 值均符合父类型的类型限定，但仍出错，因为没有找到匹配的构造方法。

实际上，目前的 `PlanePointC` 仅有一个构造方法，形如：

```
((::Type{PlanePointC})(x::T, y::T, v::V) where {T<:Signed, V<:Real})
```

其中，`Type{PlanePointC}` 结构前文已经介绍过，用于指代名为 `PlanePointC` 的参数类型；参数表后的 `where` 其实是该参数化函数的表达方式，后文会介绍。我们大致可看出其遵循了复合类型声明的限界规则，即构造方法是基于当前类型声明生成的。上例中提供的值并不符合要求，因为限定为 `Signed`，但实际是 `UInt8` 类型，所以出现了无匹配方法的错误。可见，父类型与当前参数类型中的限界规则同时起效，并不是冲突，仅是约束节点不同；可以认为，总体上遵循最严格的那个限界要求。

6.8.6 协变与逆变

事实上，Julia 还提供了另外一种参数类型具象化的方式，即协变（Covariant）与逆变（Contravariant），基本用法分别为：

```
类型名{ <: 类参值}
类型名{ >: 类参值}
```

下面我们以无限界版的参数抽象类型 `Pointy` 为例，具体解释这两者的概念与区别（其他类型相似）。

假设以协变形式对 `Pointy` 进行具象化，即：

```
julia> Pointy{<:Integer}
Pointy{#s1} where #s1<:Integer
```




可以发现，得到的是带有 `where` 关键字的具象类型。查看其内部结构，如下：

```
UnionAll
  var: TypeVar
    name: Symbol #s1
    lb: Core.TypeofBottom Union{}
    ub: Integer <: Real
  body: Pointy{#s1<:Integer} <: Any
```

可见这一具象化类型是一个典型的 `UnionAll` 结构，只不过因为未提供特定的类参符号，所以内部自动创建了 `#s1` 类型参数。实际上，我们也可以显式地使用 `where` 方式声明这样的结构，例如：

```
julia> Pointy{T} where {T<:Integer}
Pointy{T} where T<:Integer
```

可见，得到的形式与 `Pointy{<:Integer}` 完全相同；逆变情况与上述类似，不再单独示例。

在使用 `where` 关键字创建协变或逆变对象时，语法类似于：

类型名 {**T1**, **T2**, ...} **where** {**T1** <: 类参值1, **T2** >: 类参值2, ...}

也可以将多个类参分开表示，即：

类型名 {**T1**, **T2**, ...} **where** {**T1** <: 类参值1} **where** {**T2** >: 类参值2} ...

例如：

```
julia> abstract type Pointy3D{T1, T2, T3} end

julia> IntPoints = Pointy3D{T1, T2, T3} where {T1<:Signed} where {T2<:Integer}
               where {T3<:AbstractFloat};

julia> Pointy3D{Int32, UInt64, Float64} <: IntPoints
true
```

由此可见，采用协变或逆变方式对参数类型具象化时，获得的结果与前面所述大为不同。虽然提供了具体类参，但得到的仍旧是某种类似于参数类型的结构。似乎这种具象化过程仅仅是在原参数化类型的基础上打了“补丁”，进一步增加了限界规则而已。

实际上，不少语言中都存在协变与逆变机制，目的是描述一个参数类型是否沿袭了类参的继承关系：协变中，具象后的参数类型继承的父子秩序与类参是相同的；而逆变中，继承的则是相反的父子秩序。

Julia 中的沿袭规则可描述为：协变时，只有 `S<:T` 满足时下式中的继承断言才会成立：

参数类型名 {**S**} <: 参数类型名 {<:**T**}

而协变时，只有 `S>:T` 满足时下式中的继承断言才会成立：

参数类型名 {**S**} <: 参数类型名 {>:**T**}

例如：

```
julia> Pointy{Int32} <: Pointy{<:Integer}
true
```




```
julia> Pointy{Signed} <: Pointy{<:Integer}
true
```

```
julia> Pointy{Real} <: Pointy{<:Integer}
false
```

其中, `Int32` 与 `Signed` 都是 `Integer` 的子类型, 所以断言均成立; 但 `Real` 不是 `Integer` 的子类型, 所以不成立。这与前述中各具象类型之间相互独立、不存在父子关系是完全不同的, 因为内部的结构发生了变化。实际上, 如果查看某个具象类型的结构便会发现:

```
julia> dump(Pointy{Real})
Pointy{Real}
```

内部已经没有任何特殊结构, 就是一个具体的类型。对于具象的 `Point1D` 类型也是如此:

```
julia> dump(Point1D{Int64})
Point1D{Int64} <: Any
  x::Int64
```

```
julia> typeof(Point1D{Int64})
DataType
```

可见, 其已经在类型具象化的过程“蜕变”成了普通的类型; 这也是具象类型间无直接继承关系而成为独立类型的根源。

对于逆变也有相似的结论, 例如:

```
julia> Point1D{Real} <: Point1D{>:Integer}
true
```

```
julia> Point1D{Number} <: Point1D{>:Integer}
true
```

```
julia> Point1D{Int32} <: Point1D{>:Integer}
false
```

可见, 断言的成立与类参的父子秩序是相反的。

实际上, 类似于 `UnionAll`, 可以认为协变或逆变类型指代了一族类型, 其中每个类型的类参值都是协变型类参的子类型, 或是逆变型类参的父类型。换言之, 对于协变型而言, 形式 `Name{<:T}` 等效于 `Union{Name{S1}, Name{S2}, ...}`, 其中 `Si<:T` 总成立; 对于逆变型, 形式 `Name{>:T}` 等效于 `Union{Name{S1}, Name{S2}, ...}`, 其中 `Si>:T` 总成立。

6.9 常用数集

本节简单介绍 Julia 提供一些常用数集 (Collection, 或称为容器) 及使用方法。但是数集中最为常用的数组类型会在第 8 章单独阐述。

6.9.1 元组

1. 定义

元组 (Tuple) 是由多个元素构成的组合结构, 且元素的类型及个数都可不受限制; 但是一旦创建, 对象便不可变, 其中的元素类型、值及个数都无法再修改。基本的定义形式为:

```
(元素1, 元素2, ..., 元素n)
```

其中, 圆括号为界定符, 元素之间用逗号分开。例如:

```
julia> typeof((1,"foo",2.5))
Tuple{Int64,String,Float64}
```

可见 Julia 会根据元素值自动创建形如 `Tuple{T1, T2, ...}` 的元组对象。

在无歧义的情况下, 可以省略圆括号, 以逗号连接的多元素直接创建元组, 例如:

```
julia> 1,"foo",2.5
(1, "foo", 2.5)
```

```
julia> typeof(ans)
Tuple{Int64,String,Float64}
```

不过对于单元素的元素, 为避免歧义, 一般需在其尾部附加一个逗号, 即 (元素,)。

例如:

```
julia> (1,)
(1,)
```

```
julia> typeof(ans)
Tuple{Int64}
```

另外, 元组是协变的, 所以如下这种特性会在多态分发时发挥作用:

```
julia> Tuple{Int} <: Tuple{Integer}
true
```

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,Any}
true
```

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,Real}
false
```

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,}
false
```

2. 展开

创建元组对象时, 可将已有的元组通过 ... 操作符 (展开操作符) 展开到新元组中。

例如:

```
julia> (1, 2, (10.5,"abc")..., 3)
(1, 2, 10.5, "abc", 3)
```

当然, 对于某个元组变量也是适用的:

```
julia> a = (10.5, "abc")
(10.5, "abc")
```

```
julia> (1, 2, a..., 3)
(1, 2, 10.5, "abc", 3)
```

需注意的是，下面的用法是不会成功的：

```
julia> (a...)
ERROR: syntax: "... " expression outside call
```

显然，这是一个语法错误，即 Julia 并不支持这样的展开操作，正确的做法是：

```
julia> (a...,)      # 注意...之后的逗号
(10.5, "abc")
```

即需要在展开符之后再附加一个逗号。

3. 元素访问

如果要访问元组对象的元素，使用方括号并给定索引值（1-based）即可，例如：

```
julia> a[1]          # 访问第1个元素
10.5
```

```
julia> a[2]          # 访问第2个元素
"abc"
```

但内部元素不可改变：

```
julia> a[1] = 10.9
ERROR: MethodError: no method matching setindex!{::Tuple{Float64,String}, ::Float64, ::Int64}
```

例中试图修改 a 的第 1 个元素，结果报错，因为元组是不可变的。

还有一种提取元组元素的方法，即直接提供变量列表，将其中的元素“提取”到对应位置的变量中。例如：

```
julia> x, y = a
(10.5, "abc")
```

```
julia> x
10.5
```

```
julia> y
"abc"
```

提供变量的个数可以少于元组长度，但不能多，否则会报 `BoundsError` 错误。

4. 命名元组

命名元组（Named Tuple）是另外一种类型的元组，与上述的普通元组的区别在于，每个元素能够附加一个名字，基本的定义形式为：

（名1=元素1，名2=元素2，...，名n=元素n）

例如：

```
julia> nt1 = (a=1,b=2,c=3.8,d=8//9)      # 四个字段的命名元组
```

```
(a = 1, b = 2, c = 3.8, d = 8//9)
```

```
julia> typeof(nt1)
```

```
NamedTuple{(:a, :b, :c, :d), Tuple{Int64, Int64, Float64, Rational{Int64}}}
```

其中的名称与元素值可视为键值对，在定义命名元组时各个名称是不能重复的，否则会报错：

```
julia> (a=1,a=2)
```

```
ERROR: syntax: field name "a" repeated in named tuple
```

但需要注意的是，以这种方式创建命名元组的对象时，界定圆括号不再能省略，否则会出现错误。

在结构上，命名元组的类型为 `NamedTuple`，是一个参数化复合类型，包括两个部分，一部分是元素为 `Symbol` 类型的普通元组，用于记录命名元组内部各元素的名称；另一部分是记录着各元素类型的元组。例如前例中的 `nt1` 命名元组，其第一个类参为常量 `(:a, :b, :c, :d)`，是四个元素的名称元组，而另一个类参 `Tuple{Int64, Int64, Float64, Rational{Int64}}` 表示 `nt1` 的元素类型依次是两个整型 `Int64`、浮点型 `Float64` 与有理型 `Rational{Int64}`。

在构造命名元组时，自然也可以通过其构造方法创建对象，例如：

```
julia> nt2 = NamedTuple{(:a, :b), Tuple{Float32, String}}((1, ""))
```

```
(a = 1.0f0, b = "")
```

在创建对象时，类型中可同时提供名称元组与类型元组，也可只提供名称元组，由 Julia 自动确定类型。例如：

```
julia> nt3 = NamedTuple{(:a, :b)}((1, ""))
```

```
(a = 1, b = "")
```

对于命名元组对象，元素的访问除了采用普通元组的下标索引方式外，还可使用元素的名称进行访问。例如：

```
julia> nt3.a, nt3.b
(1, "")
```

```
julia> nt2.a, nt2.b
(1.0f0, "")
```

```
julia> nt1.a, nt1.b, nt1.c, nt1.d
(1, 2, 3.8, 8//9)
```

这种访问方式不但直观，而且在性能上并不弱于下标索引方式。

6.9.2 键值对

键值对 `Pair` 类型用于建立两个对象的映射 / 关联性关系，是一种参数化的不可变复合类型，一旦构造不可更新修改。其内部的基本结构是：

```
Pair{A,B} <: Any
first::A
second::B
```


该类型有两个成员，分别是 `first` 与 `second`；其中 `A` 与 `B` 是类型参数，分别控制键与值的类型。

通过构造函数可以很容易得到 `Pair` 对象，例如：

```
julia> Pair{1, 3.2}
1=>3.2

julia> typeof(ans)
Pair{Int64,Float64}

julia> Pair{Float64,Int32}(Int64(1),Int64(5))
1.0=>5

julia> typeof(ans)
Pair{Float64,Int32}
```

也可以通过特有的语法 `Key=>Value` 创建 `Pair` 对象，例如：

```
julia> a = 1=>3.2
1=>3.2

julia> b = 1.0=>5
1.0=>5

julia> c = Int32(1) => Float32(2.3)
1=>2.3f0

julia> typeof(c)
Pair{Int32,Float32}
```

对于已有的 `Pair` 对象，可以使用成员访问符获得其中的 `Key` 与 `Value` 值，例如：

```
julia> a.first
1

julia> b.second
5

julia> c.second
2.3f0
```

或者使用索引进行键或值的访问：

```
julia> a[1]      # 索引1对应键
1

julia> a[2]      # 索引2对应值
3.2

julia> a[3]      # 其他索引啥都不对应，报错
ERROR: BoundsError: attempt to access 1 => 3.2
       at index [3]
```

但若试图修改，则会报错，例如：

```
julia> a.first = 2
ERROR: type Pair is immutable
```

```
julia> a.second = 8.1
ERROR: type Pair is immutable
```

另外, Pair 的对象可以转换到元组类型, 例如:

```
julia> Tuple(b)
(1.0, 5)
```

```
julia> Tuple(a)
(1, 3.2)
```

6.9.3 字典

字典 (Dict) 是一种标准的关联型数集, 分为键集合与值集合两部分。键集合的元素要求互不相同; 值集合无特别要求。字典中的元素是无序的, 表达的是键集到值集的映射关系 (与 Pair 类同); 而且对每一个键, 值集合中有且仅有一个与之对应。

1. 定义

字典类型的原型为:

```
Dict{K,V} <: AbstractDict{K,V}
```

其中, K 为键的类型, V 为值的类型。

在内部实现中, 为保证键元素互不相同, Dict 使用 hash() 函数生成键的哈希值并采用 isequal() 函数 (与运算符 == 有差异, 可参见前文) 进行相异性判断。可以通过重载 isequal() 及 hash() 函数, 定义自己的 Dict 类型。

使用构造函数创建 Dict 对象, 只需将多个 Pair 对象作为参数即可, 例如:

```
julia> Dict{1=>1.1, 2=>2.2}
Dict{Int64,Float64} with 2 entries:      # 自动推断Dict的类型参数
  2 => 2.2
  1 => 1.1

julia> Dict{1=>1.1, 2.2=>2}
Dict{Any,Any} with 2 entries:
  2.2 => 2
  1   => 1.1
```

Julia 会自动推断键与值的类型。当然, 我们也可以显式地限定 Dict 的类型, 例如:

```
julia> Dict{Int32,Float32}(1=>1.1, 2=>2.2)
Dict{Int32,Float32} with 2 entries:
  2 => 2.2
  1 => 1.1

julia> Dict{Real,Real}(1=>1.1, 2.2=>2)
Dict{Real,Real} with 2 entries:
  2.2 => 2
  1   => 1.1
```

对于有连续性特点的元素, 可结合遍历表达式及函数创建 Dict 对象, 例如:

```
julia> function f(x)                                # 函数的定义后文会介绍
    x^2
```

```

        end
    f (generic function with 1 method)

julia> z = Dict{i=>f(i) for i = 1:5}           # 遍历表达式（后文介绍）
Dict{Int64,Int64} with 5 entries:
  4 => 16
  2 =>  4
  3 =>  9
  5 => 25
  1 =>  1

```

或者:

```

julia> z = Dict{i=>i^2 for i in 1:5}          # 匿名函数后文会介绍
Dict{Int64,Int64} with 5 entries:
  4 => 16
  2 =>  4
  3 =>  9
  5 => 25
  1 =>  1

```

也可以将已有的元组结构转换到 Dict 对象, 例如:

```

julia> a = (1=>1.1, 2=>2.2)
(1=>1.1, 2=>2.2)

julia> x = Dict{a}
Dict{Int64,Float64} with 2 entries:
  2 => 2.2
  1 => 1.1

```

数组结构同样也可以:

```

julia> b = [1=>1.1, 2=>2.2]
2-element Array{Pair{Int64,Float64},1}:
 1=>1.1
 2=>2.2

julia> y = Dict{b}
Dict{Int64,Float64} with 2 entries:
  2 => 2.2
  1 => 1.1

```

2. 元素访问

Dict 对象中的元素访问可使用中括号的索引方式 (索引值为键值) 进行, 例如:

```

julia> z[4]                                     # 键4与值16对应
16

julia> z[2]                                     # 键2与值4对应
4

julia> z[5]                                     # 键5与值25对应
25

```

再例如:

```

julia> p = Dict{"A"=>1,"B"=>2}
Dict{String,Int64} with 2 entries:

```

```
"B" => 2
"A" => 1
```

```
julia> p["A"]
1
```

```
julia> p["B"]
2
```

如果索引的键不存在，可能会报错，例如：

```
julia> p["C"]
ERROR: KeyError: key "C" not found
```

但若提取 Dict 元素时直接赋值，键不存在时不会报错，而是直接将赋值中的对应关系新增到 Dict 对象中，例如：

```
julia> p["C"] = 4
4
```

```
julia> p
Dict{String,Int64} with 3 entries:
  "B" => 2
  "A" => 1
  "C" => 4
```

若原对应关系已经存在，则会改变原有的键值对应关系，例如：

```
julia> p["C"] = 3
3
```

```
julia> p
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
  "C" => 3
```

元素访问还可使用 get() 函数，好处是当元素不存在时能以指定的默认值返回，例如：

```
julia> get(p, "A", 1) # 取得"A"的对应值，因为存在该键，所以返回原对应值3
3
```

```
julia> get(p, "D", 5) # 取得"D"的对应值，因为该键不存在，所以返回默认值5
5
```

若希望键不存在时将默认值存入 Dict 中，则可使用 get!() 函数，例如：

```
julia> get!(p, "D", 5) # 虽然"D"不存在，但同get()一样会返回默认值5
5
```

```
julia> p # 但经过get!()函数调用，关系"D"=>5已经存入了字典p中
Dict{String,Int64} with 4 entries:
  "B" => 2
  "A" => 3
  "C" => 4
  "D" => 5
```

```
julia> get!(p, "D", 1) # "D"已经存在，直接返回内部的值而非默认值
5
```


如果发现某个键多余, 可使用 `delete!()` 函数将其删除, 例如:

```
julia> delete!(p, "D")
Dict{String,Int64} with 3 entries:
  "B" => 2
  "A" => 3
  "C" => 4
```

该函数会在执行后返回修改后的 `Dict` 对象。

如果希望在删除某映射时同时取出, 使用 `pop!()` 函数即可:

```
julia> pop!(p, "C")
4
julia> p
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 3
```

该函数在删除指定键的同时, 会将该键对应的值返回。

3. 遍历

可通过 `length()` 函数或其成员字段 `count` 获知 `Dict` 中元素的个数, 例如:

```
julia> length(p)
2
julia> p.count
2
```

若要遍历 `Dict` 所有元素, 一个直接的方式是通过 `keys()` 函数取得其键集合。例如:

```
julia> p
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 3
julia> keys(p)
Base.KeySet for a Dict{String,Int64} with 2 entries. Keys:
  "B"
  "A"
```

该函数会生成键集的迭代器, 然后便可使用 `for` 循环进行遍历:

```
julia> for k in keys(p)
    println(k)
end
B
A
```

这样, 通过键的遍历就可访问 `Dict` 中的所有元素了, 即:

```
julia> for k in keys(p)
    println(k, "'s value is ", p[k])
end
B's value is 2
A's value is 3
```

同样, 可使用 `values()` 函数取得值集合并可直接遍历, 例如:

```
julia> values(p)
Base.ValueIterator for a Dict{String,Int64} with 2 entries. Values:
 2
 3

julia> for v in values(p)
    println(v)
end
2
3
```

如果希望将该函数获得的迭代器转为数组使用，对其调用 `collect()` 函数即可，即：

```
julia> collect(values(p))
2-element Array{Int64,1}:
 2
 3
```

6.9.4 集合

Julia 中的集合类型 `Set` 等效于数学中的集合定义，其中的元素互不相同。原型为：

```
Set{T} <: AbstractSet{T}
dict::Dict{T,Nothing}
```

可见该类型以 `Dict` 为基础，也是无序的。

实例化时不能直接将元素单个以参数的形式提供给构造函数，而需要以可迭代数集的方式提供，如元组、数组或迭代器等。例如：

```
julia> a = Set([1, 2, 2, 3, 3, 4, 4]) # 通过数组构造，重复元素被忽略
Set{Int64}([1, 2, 3, 4])
```

```
julia> b = Set{(1, 2, 3, 4)} # 通过元组构造
Set{Tuple{Int64, Int64, Int64, Int64}}([1, 2, 3, 4])
```

```
julia> p
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 3
```

```
julia> c = Set(keys(p)) # 借用上节中Dict键集合的迭代器
Set{String}(["B", "A"])
```

可见，如果输入的迭代数集中元素重复，则会被 `Set` 自动忽略而仅保留一个。

需要注意的是，无法通过索引的方式对 `Set` 进行元素访问，只能以迭代遍历的方式进行访问：

```
julia> for i in a
    print(i, " ")
end
4 2 3 1
```

可以使用 `in` 作为操作符判断某个元素是否在 `Set` 中，例如：

```
julia> 4 in a
true
```

```
julia> 8 in a
false
```

对 Set 对象可以进行经典的集合运算，包括求交集、并集、补集等，例如：

```
julia> a = Set([1,2,3,4])
Set{Int64}([4, 2, 3, 1])
```

```
julia> b = Set([1,3,5])
Set{Int64}([3, 5, 1])
```

```
julia> x = union(a,b)      # 并集
Set{Int64}([4, 2, 3, 5, 1])
```

```
julia> y = intersect(a,b)  # 交集
Set{Int64}([3, 1])
```

```
julia> z = setdiff(a,b)    # 补集，在a中，但不在b中
Set{Int64}([4, 2])
```

当然，也支持集合的包含关系判断，例如：

```
julia> issubset(b, a)      # a是否包含b，即b是否是a的子集
false
```

```
julia> issubset(b, b)      # b包含b自身
true
```

```
julia> issubset(z, a)      # a是否包含z
true
```

甚至可以基于 Unicode 的支持，直接使用数学中的集合运算符号，即：

```
julia> b ⊆ a
false
```

```
julia> b ⊆ b
true
```

```
julia> z ⊆ a
true
```

这样看起来更为直观自然。

如果要判断一个集合是否是空集，可使用 `isempty()` 函数：

```
julia> isempty(a)
false
```

该函数实际适用于 Julia 中的各种数集，包括元组、数组、字典等。

如果要清除集合中的所有元素，直接调用 `empty!()` 函数即可：

```
julia> empty!(a)
Set{Int64}()
```



注意 函数名存在标识符！（感叹号），表示会修改输入的数据，所以一旦调用，对象中的所有元素将会消失，使用时请小心。

6.10 缺失值的表达

6.10.1 missing

在数理统计、数据挖掘或分析、机器学习等需要大规模数据的应用场景中，经常会出现缺失值（理论上应该存在，但观测数据没有实际的值），Julia 提供了专门的 Missing 类型用于表达这种数据，而 missing 是该类型仅有的单例对象。该对象的作用类似于 R 语言中的 NA，或 SQL 中的 NULL。

不过，在使用 missing 中，不论是数学运算符还是函数，缺失性都会进行自动地传播。这样做的合理性很容易理解：输入值的不确定必然导致输出值的不确定，即涉及缺失值的操作一般均会生成缺失值。下面举例说明：

```
julia> missing + 1
missing
```

```
julia> "a" * missing
missing
```

```
julia> abs(missing)
missing
```

missing 是 Julia 的正常对象，但只有当函数对其做出恰当的兼容性处理才能正常工作。第三方包同样需要对 Missing 类型做出恰当的处理，例如在定义函数时，如果实参是 missing 而定义的函数没有恰当处理，则会抛出 MethodError 异常。

missing 对象在参与逻辑运算时，同样会进行缺失性传播，一旦有操作数是 missing 对象，那么结果也将是 missing，例如：

```
julia> missing == 1
missing
```

```
julia> missing == missing
missing
```

```
julia> missing < 1
missing
```

```
julia> 2 >= missing
missing
```

需要注意的是，是否相等运算符 == 也是适用传播规则的，所以该运算符无法用于检测一个值是否是 missing。如果需要进行检测，可使用 ismissing() 函数。不过，函数 isequal() 和是否相同运算符 === 是这种传播规则的特例：即使参数或操作数存在缺失值，仍会正常地返回一个布尔值结果。所以，这两个操作也可以用于测试某个值是否是 missing 值，例如：

```
julia> ismissing(missing)
true
```

```
julia> missing === 1
```



```
false
```

```
julia> isequal(missing, 1)
false
```

```
julia> missing === missing
true
```

```
julia> isequal(missing, missing)
true
```

对于大小比较, 虽然运算符受到传播规则的影响无法返回正常的布尔值, 不过可以使用函数 `isless()`, 这是排序方法 `sort()` 的基础函数。例如:

```
julia> isless(1, missing)
true
```

```
julia> isless(missing, Inf)
false
```

```
julia> isless(missing, missing)
false
```

这是因为在 Julia 的数值系统中, `missing` 值被处理为比任意类型的数值都大。

逻辑运算符在缺失传播方面同样有些特殊, 是否能够产生正常布尔结果依赖于逻辑操作的定义。例如:

```
julia> true | missing
true
```

```
julia> missing | true
true
```

对于或操作, 操作数中只要任意一个是 `true`, 则无论其他几个是否为 `true`, 结果都会是 `true`, 所以 `missing` 即使作为 `true` 之外的操作数并不会影响到结果的正常输出。

同样, 对于与操作, 操作数有一个是 `false` 的情况, 无论其他操作数是否存在 `missing` 值, 结果都会正常地输出 `false` 值, 即:

```
julia> missing & false
false
```

```
julia> false & missing
false
```

反之, 缺失性会传递。在或操作中, 如果其中一个操作数是 `false`, 那么另外操作数是 `missing` 时, 导致结果只能取 `missing`, 即缺失性传递下去了:

```
julia> false | missing
missing
```

```
julia> missing | false
missing
```

同时, 在与操作中, 其中一个操作数为 `true` 时则传递缺失性, 例如:

```
julia> true & missing
missing
```

```
julia> missing & true
missing
```

另外，缺失性总会在异或操作中传播，这是因为在这种操作中，两个操作数都会对结果产生影响。同样，反操作符 `!` 在操作数为 `missing` 时也总会返回 `missing` 值。

在 `if`、`while` 及 `?:` 等控制逻辑结构中，条件表达式中是不允许出现 `missing` 的。这是因为 `missing` 无法明确地告知条件是否满足，所以 Julia 也无从判断后续的代码是否应该执行或者该如何执行，所以这种情况下会抛出 `TypeError` 异常。

在 `&&` 及 `||` 的运算中，`missing` 的出现通常会引发异常，即：

```
julia> missing || false
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

```
julia> missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

```
julia> true && missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

与前述的逻辑运算类似的是，仅在 `missing` 是否存在都不会影响到判断时才能够正确输出结果（通常 `missing` 作为最后一个操作数）：

```
julia> true || missing
true
```

```
julia> false && missing
false
```

6.10.2 nothing

Julia 中有一个 `Nothing` 类型，用于表示什么都没有，即不需要在内存中表达的一类事物。而常量 `nothing` 是其唯一的实例，即单例对象。

`nothing` 的存在意义仅因为语法描述的便利，表示不运行也不输出任何东西，可以在一些语法结构中作为占位使用。例如，一些函数在不需要返回值时，便可在最后附加一个 `nothing` 语句，或者在类似 `if false; end` 这样的语言结构中也会输出 `nothing`。

在 REPL 中，因为对象为 `nothing`，所以输出时什么都不会打印。

6.10.3 可有可无的表达

之前介绍过了 `NaN`、`nothing` 及 `missing` 这三个特殊值[⊖]：`NaN` 是浮点型常量，表示非数值对象，属于数值层面的概念；`nothing` 类型的单例常量，表示什么都没有，是语法层面的定义；`missing` 则表示数据区某个字段值的缺失，是数据的一种状态。

[⊖] Julia v0.4 之前的版本还有个 `None`，但已被弃用，由 `Union{}` 取代。

但有一种场景这三者都不能合理表述：例如某个名为 x 的变量，类型为 T ，在处理过程中，其值不断地变化的；很有可能在某个环节出现了空值（缺失）的情况，比如从外部文件加载数据时，因为数据无效导致值无法提供， x 只能置空；但又不总是出现，也有值可用的情况，这种状况在大规模数据中经常会遇到。

但此时 x 的表达就出现了困难：一则未必总是浮点型，二则顺利的情况下也会有确切的值。为了能够表达这种“可有可无”的情况，可以 `Union{T, Nothing}` 进行表达。这种方式等效于其他语言中的 `Nullable`、`Option` 或 `Maybe` 类型，而且这种类型联合结构能够用于函数的参数，复合类型的成员字段，甚至是数组的元素类型。

如果值本身的正常取值是 `nothing`，但也存在缺失值，则可以使用 `Union{Some{T}, Nothing}` 表达。当 `x==nothing` 时，表示值缺失；而当 `x==Some(nothing)` 时，则表示该值存在，只不过取值为 `nothing` 对象。在处理中，类似于 SQL 中的 `COALESCE` 函数，Julia 的 `something()` 函数可以返回参数表中第一个不是 `nothing` 的值。例如：

```
julia> something(nothing, 1)
1

julia> something(Some(1), nothing)
1

julia> something(missing, nothing)
missing

julia> something(nothing, nothing)
ERROR: ArgumentError: No value arguments present
```

不过如果参数中不存在 `nothing` 之外的有效值，该函数会抛出 `ArgumentError` 异常。对于某个 `Some` 对象，可以使用 `.value` 获得其内部有效值。

除此之外，在 `Nullable.jl` 包中，提供了一种自 v6.0 版本中抽离出来的单独维护的 `Nullable` 类型，也可以处理这种可有可无的情况。其中的 `Nullable` 类型，并不表示具体的空值，而是某个值存在可能为空、也可能不空的两可情况。其结构为：

```
Nullable{T} <: Any
  hasvalue::Bool
  value::T
```

其中， T 是类型参数；`hasvalue` 标记其是否有值存储，而 `value` 则记录着 T 类型的数据内容。

若要使用该类型，只需调用其构造函数便可创建一个 `Nullable` 对象，例如：

```
julia> Nullable{Int64}()      # 空的情况，即这个Int64对象里没内容
Nullable{Int64}()

julia> Nullable{Tuple}()     # 空的情况，即这个Tuple对象里没内容
Nullable{Tuple}()
```

或者是自定义的复合类型：

```
julia> struct MyType
    x
```



```

y
end

```

```

julia> Nullable{MyType}() # 空的情况，即这个MyType对象里没内容
Nullable{MyType}()

```

当然可以将已有的值封装进去：

```

julia> a = MyType(1,2,3)
MyType(1, 2,3)

julia> m = Nullable(1)
Nullable{Int64}(1)

julia> n = Nullable((1,2,3))
Nullable{Tuple{Int64,Int64,Int64}}((1, 2, 3))

julia> p = Nullable(a)
Nullable{MyType}(MyType(1, 2,3))

```

也支持 NaN 与 nothing 这两个特殊值，即：

```

julia> q = Nullable(NaN)
Nullable{Float64}(NaN)

julia> r = Nullable(nothing)
Nullable{Nothing}(nothing)

```

对于 Nullable 对象，随时可以使用 isnull() 函数判断其状态。如果无值，该函数会返回 true，否则返回 false，例如：

```

julia> isnull(Nullable{Int64}())
true

julia> isnull(Nullable{Tuple}())
true

```

```

julia> isnull(m)
false

```

```

julia> isnull(p)
false

```



注意 有个特殊情况需要注意，即：

```

julia> isnull(Nullable(NaN)) # Float64类型，有内容，但为NaN
false

julia> isnull(Nullable(nothing)) # Void类型，有内容，但为nothing
false

```

从此例可对 Nullable 作进一步理解：Nullable 表达的仅仅是封装类型在物理上是否有内容；至于内容本身的意义，Nullable 并不关心。

对于无元素的元组、数组或其他数集类型，Nullable 同样不会判定为无内容，即：


```
julia> isnull( Nullable{ () } )
false
```

```
julia> isnull( Nullable{ [] } )
false
```

因为虽然无元素，但数集对象是确实存在的。

一旦发现某个 Nullable 对象是有内容的，便可使用 get() 函数取得该内容：

```
julia> get(m)
1
```

```
julia> get(p)
MyType(1, 2.3)
```

```
julia> get(q)
NaN
```

```
julia> get(r)
# 空白
```

取的是nothing，但正如前所述，什么都不打印

但若实际并无内容，会报出异常，例如：

```
julia> get(Nullable{Float64}())
ERROR:NullException()
```

这样能够确保调试运行期间及时发现问题，并可通过异常机制进行妥善处理。

如果应用中 Nullable 封装的数据不是关键内容，有则用，无则遵循惯例，所以在使用函数 get() 时可提供默认值，这样的话会更为便利。例如：

```
julia> get(Nullable{Float64}(), 0.0) # 为空，取0.0
0.0
```

```
julia> get(Nullable(1.0), 0.0) # 不为空，取实际值，忽略默认值
1.0
```



注意 默认值的类型必须与 Nullable 封装的类型一致，否则会导致结果不可靠甚至引发错误。

参数化类型 Nullable 虽然可在数据处理层面提供一种安全机制，但不能表达语法意义上的无效情况。例如，某些变量、字段或元素没有初始化便被访问，程序会立即抛出异常，但不能通过 Nullable 的封装来解决问题，而需要用 isdefined() 函数预先测试以便进一步处理。

此外，Julia 的 Base 模块中还有一个 Ptr{Nothing} 类型的常量 C_NULL，用于混合编程时兼容 C 语言的 NULL 指针，概念也与 Nullable 不同。此处不做过多介绍。

函 数

函数是逻辑实现的核心结构，也是功能划分的主要方式。在“一切皆对象”的 Julia 中，函数不仅是处理过程或表达式语句的集合，同样也是对象的一种，有着自己的类型。函数在 Julia 中也是多态机制实现的主要载体。设计者可以定义多个同名函数，构成一套实现方法，为不同的参数组合提供统一的接口。

本章详细介绍函数的定义、调用与参数化方法，还有以其为基础的动态分发机制。另外，由于 Julia 中复合类型的构造函数一般在结构的外部实现，所以也会在本章深入介绍。

7.1 基本定义

7.1.1 常规结构

函数的常规定义语法为：

```
function 函数名(参变量1, 参变量2, ...)  
    # 实现语句  
    return 结果表达式  
end
```

其中，关键字 `function` 与 `end` 界定了函数的定义范围。

参数变量（参变量）用于控制函数的行为，需在紧随函数名之后（中间不能有空格）的圆括号内给出，可以使用 `::` 操作符进行类型限定。函数定义可以不提供参数，即参变量表为空，但圆括号不能省略。参变量表可以看作有秩序的（positional）元组结构；在函数调用时，实际参数（实参）值需要按照参数的顺序逐一对应提供，否则会带来不可预料的结果。

函数内部的实现语句相当于复合表达式，默认最后一个表达式的计算结果作为函数的返回结果；也可以显式地使用 `return` 关键字在函数内的任意位置立即将结果返回，不再

执行后续的语句，可结合控制逻辑使用，以避免冗余的处理过程，提高效率；如果不需要结果返回，则可以在返回的地方（如 `end` 之前）提供 `nothing` 对象，即什么都不需要。

另外还有一种简洁的函数定义方式，即将复合表达式直接赋值给待定义的函数，语法为：

函数名(参变量1, 参变量2, ...) = 复合表达式

这种方式适合于比较简短的函数定义。

如果是在 REPL 中定义函数，回车后便会看到类似以下的提示信息：

函数名 (generic function with 1 method)

表明函数创建成功，之后便可以调用该函数了。在 Julia 中，函数一般被称为“generic function”，并会提示“with n methods”，其中的数字 n 表示同名函数有几种实现（后文介绍）。

函数的调用方式并不复杂，只需在函数名后提供实参值即可，形式为：

函数名(实参1, 实参2, ...)

其中的实参需是有效的、存在的对象。在调用处可以直接将返回结果赋值给某个变量或用于修改某个数据区。

下面以实例说明函数的定义方法。例如，需要一个累加两个值的函数：

```
julia> function addtwo(x, y)
    x + y
end
addtwo (generic function with 1 method)
```

因为简单，所以可以采用简洁的方式，如下：

```
julia> addtwo(x, y) = x + y
addtwo (generic function with 1 method)
```

尝试调用一下：

```
julia> addtwo(5, 10)
15
```

但如果加法语句之间存在 `return` 语句且被执行，则不会得到预期的结果，例如：

```
julia> function notaddtwo(x, y)
    return x * y
    x + y
end
notaddtwo (generic function with 1 method)
```

```
julia> notaddtwo(2, 3)
```

```
6
```

该函数在计算 `x*y` 之后便立即将乘积返回，后续的加法语句并没有执行。

7.1.2 类型限定

Julia 是弱类型语言，使用变量或参数是可以不限类型，但类型的自由选择往往会出现非常意外的结果。例如：

```
julia> f(x,y) = 2x + y
f (generic function with 1 method)
```

对于数值类型，能顺利得到想要的结果：

```
julia> f(2, 3.0)
7.0
```

但如果调用不当，传入了非数值类型的参数，则会出现奇怪的结果：

```
julia> f1(3, 'a')
'g': ASCII/Unicode U+0067 (category Ll: Letter, lowercase)
```

所以，在程序开发过程中，尤其是关键计算处，最好都能够明确地限定类型，以免给自己“挖坑”。所谓“坑”，是那些不会报错但还可以顺利执行的代码，而结果却不是预期的，这也是最难以发现的问题。

若要在函数定义时进行类型限定，只需在参变量后使用 `::` 操作符给出其类型条件即可。基本方式为：

```
function 函数名(参变量1::类型1, 参变量2::类型2, ...)
    # 实现
end
```

或者

```
函数名(参变量1::类型1, 参变量2::类型2, ...) = 复合表达式
```

其中的类型可以是具体类型也可以是抽象类型；也可以只限定部分参数或全部参数。

例如，实现的 `addtwo()` 函数如果只支持 `Int64` 类型，则可以将其定义修改为：

```
julia> function addtwo(x::Int64, y::Int64)
    x + y
end
```

当输入参数符合要求时，可以顺序执行：

```
julia> addtwo(Int64(10), Int64(5))
15
```

但如果输入参数不符合要求，则会报错，例如：

```
julia> addtwo(Int32(10), Int32(5))
ERROR: MethodError: no method matching addtwo(::Int32, ::Int32)

julia> addtwo(Float64(10.0), Int64(5))
ERROR: MethodError: no method matching addtwo(::Float64, ::Int64)
```

这样，在函数内部执行前我们便能够知道发生了问题，可在调用处进行恰当的调整。

如果希望该函数能够支持所有的整型，则可采用抽象类型 `Integer` 来约束参数，以放大其适用的范围，重新定义如下：

```
julia> function addtwo(x::Integer, y::Integer)
    x + y
end
```

此时再次提供 `Int32` 类型的参数，便能顺序地执行：


```
julia> addtwo(Int32(10), Int32(5))
15
```

同时也支持了其他的整型参数，例如：

```
julia> addtwo(Int32(10), Int8(5))
15
```

```
julia> addtwo(UInt32(10), Int8(5))
15
```

但对于非整型的数值仍然是不支持的，依然会报错。

如上所述，在定义函数时，可以通过 `::` 操作符对参变量进行类型限定。显然，若限定为 `Any` 类型是毫无意义的；但若限定的都是元类型，兼容性会变差。所以在类型限时，需要根据实际需求，选择恰当的类型节点或抽象层次对参数类型做出合理的限定。

为了叙述的方便，后续在不必要的情况会省略类型限制语法。

7.1.3 共享传参

鉴于性能问题，为避免参数传递时带来大量的内存操作消耗，Julia 采用一种被称为共享传参（pass-by-sharing）的方式：传递参数时，元类型作为值传递而可变的复合类型则按引用传递。这种方式综合了按值传参（pass-by-value）和按引用传参（pass-by-reference）的优点。

先看简单的元类型参数，例如：

```
julia> x = 10;

julia> function change_value(y)
    y = 17
end
change_value (generic function with 1 method)

julia> change_value(x)
17
```

该函数希望能改变传入参数 `x` 的值，但实际上变量 `x` 的值并没有发生变化：

```
julia> x
10          # 仍是10
```

在上例中，变量 `x` 绑定了一个 `Int64` 类型的值，因为是元类型，所以会按值传入 `change_value()` 中。此时，参变量 `y` 被创建为新的 `Int64` 对象，并被赋值（复制）为 `x` 的内容，即 `y` 被绑定到了新的内存区且不同于 `x`。在执行 `y=17` 语句时，`y` 又被重新绑定到了新的值。所以自 `x` 内容在传参时被复制到 `y` 后，整个处理过程都是针对新的内容区，与实参 `x` 的关系已经不大。

但如果实参的内部结构复杂（如复合类型），便会按引用传递。此时函数内部若是改变了参变量原有的数据，变化便会传递到函数外部的实参，也会影响原数据在外部后续处理过程中的使用。例如：

```
julia> function change_dict(a::Dict)
    a[88] = 6.6
    # 将键88的值修改为6.6
```

```

        end
change_dict (generic function with 1 method)

julia> d = Dict{88=>8.8, 99=>9.9}      # 创建Dict对象, 键88对应8.8
Dict{Int64,Float64} with 2 entries:
  99 => 9.9
  88 => 8.8

julia> change_dict(d);

julia> d
Dict{Int64,Float64} with 2 entries:
  99 => 9.9
  88 => 6.6                      # 值已被更新!

```

其中 Dict 对象先被绑定到 d 变量, 在调用时再被绑定到参变量 a。因为 a 只不过是指向 Dict 对象的新引用, 所以与 d 共享着同一个对象, 函数内部对 a 的变更会反映到外部 d 变量中。上述这种区别是非常重要的, 需要开发者注意。



提示 为了能让这种改变参数的函数行为更为醒目, Julia 建议使用感叹号! 标识这种函数。所以 change_dict 最好命名为 change_dict! 形式。这不是强制的做法, 仅希望以惯例的形式提醒开发者和调用者, 该函数内部更新了控制参数。

7.1.4 数集展开式调用

如前所述, 函数的参数表可视为元组结构, 所以在提供实参时, 我们可以直接提供元组对象。但为了各参变量能获得对应的实参, 在以元组方式提供实参时, 需附加 ... 操作符 (展开操作符, 参见 6.9.1 节), 以便将元组自动展开并提取到对应的参变量中。例如:

```

julia> addtwo((1,2)...)           # x与y分别在元组展开时获得1与2
3

julia> addtwo(1, (2,)...)         # x=1, 但y在(2,)展开时获得2
3

```

其中, addtwo() 函数有两个参变量 x 与 y, 均是在元组结构展开过程中提取了对应的实参值。

更为方便的是, 基于展开操作符的实参提供方式, 同样适用于其他可迭代数集, 包括数组 (后文介绍)、Set 等。例如:

```

julia> addtwo([1,2]...)
3

julia> s = Set{([1,2])}
Set{([2, 1])}                      # 顺序与输入时并不一致

julia> addtwo(s...)
3

julia> addtwo(1:2...)
3

```



注意 Set 这种容器是无序的，虽然能正常使用，但无法保证与有序参数的对应关系，所以仍是建议采用元组或数组的方式。

使用展开方式时，提供的常规实参与元组元素的总数量必须与有实参需求的参变量数目一致，否则会出错。另外，如果使用了数集作为参数但没有采用展开式表达，则该数集会被视为普通实参，只会被传递到其中一个参变量中，不会再被提取到其他参变量中。例如：

```
julia> addtwo((1,2))  
ERROR: MethodError: no method matching addtwo(::Tuple{Int64,Int64})
```

其中的元组仅是普通的实参，但该函数并不接收一个为元组的参数，所以报错。



注意 采用数集展开提供实参时，必须满足函数参数对类型的要求；而且因为展开作用，数集已被提取到基本参变量中，所以对作为实参的数集不再是引用传递；但是如果数集内部被提取的元素是复杂结构，仍存在引用关系。

7.1.5 多返回值

在 Julia 的函数中同时返回多个结果是比较简单的：将需要的结果以逗号方式连接，便能够以元组结构的形式同时将它们返回。例如：

```
julia> function addmul(a, b)  
    a+b, a*b  
end
```

调用后，便可以元组的方式提取结果：

```
julia> x, y = addmul(5, 10)  
(15, 50)  
  
julia> result = addmul(5,10)  
(15, 50)
```

即：

```
julia> result[1] == x == 15  
true
```

显然，两种方式获得的结果是一致的。

如果有必要，在结果返回处，可使用圆括号显式地以元组形式对多个结果进行封装。

7.2 参数传递方式

7.2.1 默认参数

如果有些参数遵循某种惯例，或者只在特定的场景下才会使用，则可以为参数提供可选值 (Optional) 或默认值。基本语法为：


```
function 函数(参数1, 参数2, ... 参数m=默认值m, 参数n=默认值n)
    # 实现体
end
```

或

```
函数(参数1, 参数2, ... 参数m=默认值, 参数n=默认值) = 复合表达式
```

例如函数 `mouse_move(x, y, speed)` 控制着鼠标的移动, 其中, 参数 `x` 和 `y` 为目的位置, 参数 `speed` 控制着移动速度。如果 `speed` 有惯用的取值, 则可以默认参数的形式提供:

```
function mouse_move(x, y, speed=50)
    #
end
```

该函数调用时, 便可以不提供 `speed` 参数, 例如:

```
julia> mouse_move(40, 120);           # 以50的速度移动到坐标(40, 120)
```

当然, 也可以提供 `speed` 值, 以替换掉默认参数值:

```
julia> mouse_move(40, 120, 90);      # 以90的速度移动到坐标(40, 120)
```

显而易见, 也可为所有的参数提供默认值, 下面仍以鼠标移动函数为例:

```
function mouse_move(x=100, y=200, speed=50)
    #
end
```

可以如下使用:

```
julia> mouse_move();                 # 以50的速度移动到坐标(100, 200)
```

```
julia> mouse_move(300);              # 以50的速度移动到坐标(300, 200)
```

可见此时提供的实参会按序传递到参变量中。

默认参数机制在很多时候能够为开发者提供很大的方便, 在代码重构、改造或进行兼容性设计时, 也会大有用处。

但由于参数表是有序的, 所以使用默认参数也是有约束的。在定义中, 有默认值的参数只能放在参数表的尾部, 而且有默认值和无默认值的参数不能混杂。因为无默认值的参数必须在函数调用时优先提供, 所以只能放在参数表前部。如果在任意默认参数后出现了无默认值的参数, 函数定义时便不会成功, 并上报语法错误, 提示 “optional positional arguments must occur at end”。

7.2.2 键值参数

对于控制逻辑复杂的函数, 往往需要大量的输入参数。此时的参数列表一般很长, 维护起来并不容易。尤其是对这种函数调用时, 因为参数表有序, 所以只能按序提供输入值, 偶尔出现的乱序问题经常会给开发带来不小麻烦。即便在定义中尽可能多地提供默认值, 但仍然治标不治本。



典型的例子是图表绘制的 `plot` 函数，其中可能需要线型、宽度、颜色、箭头形式、图例位置、坐标轴情况等等繁杂的控制参数。在调用时，不同的图表会需要配置不同的参数并忽略其他参数；但因为默认参数必须在参数表尾部，所以无法无序地跳跃设定仅需的参数。而且参数列表很长也会导致调用处的实参值列表变得很长，在修改维护时很难区分与参变量的对应关系。

如果传参时带有参数的名字，便可更为直观地知道提供的是哪个参数值；如果能够借鉴前面介绍的字典（参见 6.9 节）的结构，便可让参数表具备无序性和直观性的特点，即按键名跳着配置仅需的参数。Julia 提供了这种“键值对”设置参数的方式，此时函数头的一般形式为：

```
函数名(有序参数表; 键名1=值1, 键名2=值2, ...)
```

或可同时限定类型：

```
函数名(有序参数表; 键名1::类型1=值1, 键名2::类型2=值2, ...)
```

其中，“键名 i = 值 i ”便是键值参数（Keyword Arguments），以“键名 i ”为参数名，“值 i ”是该参数的默认值。这种键值参数的特性类似于 `Pair` 类型，显然，默认值是不可少的，必须提供。

函数定义时，键值参数表需在已有的有序参数表之后，两者间需以英文分号区分。有序参数表可以没有，但分割用的分号不可省略，即：

```
函数名(; 键名1=值1, 键名2=值2, ...)
```

这是因为如果分号省略，键值部分便成了带有默认值的有序参数表了。

对使用了键值参数的函数进行调用，有两种可选方式：

```
函数名(有序实参表, 键名1=值1, 键名2=值2, ...)    # 有序实参表后为逗号
函数名(有序实参表; 键名1=值1, 键名2=值2, ...)    # 有序实参表后为分号
```

显然，键值对是无序的，而且可以在调用时不提供不需要的参数。但提供的参数必须同时带有键名，即需以 `Pair` 形式配置。同样，若有序实参表缺失，则只能采用分号形式且分号不可少。如果同时存在可变参数（下文介绍），也只能使用分号形式。

另外，调用时也可采用 `Pair` 方式提供键值参数，但键名（参数名）需以 `Symbol` 类型输入，即在名字前加冒号：标识符（后文介绍），即：

```
函数名(有序实参表; :键名1=>值1, :键名2=>值2, ...)
```

注意，此时必须使用分号形式。

下面，我们仍以绘图 `plot()` 函数为例，说明键值参数的用法。定义如下：

```
julia> function plot(x, y; style="solid", width=1, color="black")
    print("ok")
end
plot (generic function with 1 method)
```

提供所有的参数进行调用，代码如下：

```
julia> plot(1, 2, width=30, style="solid", color="black")    # 逗号分隔方式
ok
```



```
# Pair方式, 且未按顺序提供键值参数的值
julia> plot(1, 2; :color=>"black", :style=>"solid", :width=>30)
ok
```

其中提供的键值参数并没有遵循定义的顺序。或者, 完全不提供键值实参, 例如:

```
julia> plot(1, 2)
ok
```

当然, 只提供部分键值参数也是可以的:

```
julia> plot(1, 2, width=30)
ok

julia> plot(1, 2, color="yellow")
ok

julia> plot(1, 2, style="solid", color="yellow")
ok
```

所以, 提供键值参数时不必依序, 可任意地选择某个感兴趣的键值参数进行配置。

需要注意的是, 在传递的键值实参列表中如果存在重复键值对, 会抛出异常。例如:

```
julia> plot(1, 2, width=30, width=40)
ERROR: syntax: keyword argument "width" repeated in call to "plot"
```

还有一种情况, 在函数定义时, 键值参数使用了表达式, 而且后一键值的默认值表达式中出现了前面的键名。此时, 在该默认值计算时, 会将表达中出现的键名作变量处理, 且各变量的取值为对应的默认值。例如:

```
julia> function bar(; x=1, y=x+3, z=x+y)
    println("x=", x)
    println("y=", y)
    println("z=", z)
end
bar (generic function with 1 method)

julia> bar(; x=1)           # y默认值由x+3计算出, z则有x与计算后的y累加
x=1
y=4
z=5

julia> bar(; x=1, z=10)    # 因z提供了值, 故其默认计算的结果被替代
x=1
y=4
z=10
```

7.2.3 可变参数

在一些应用场景中, 希望定义的函数能够接收任意数量的参数, 例如经典的打印输出函数或字符串格式化函数。如果参数个数固定, 在使用这种函数时, 只能反复地调用, 并需要对结果进行无谓地拼装。

为此, 可变参数 (Varargs) 便成为此时最适合的解决的方案。在 Julia 中, 提供了较为直接、简便的支持, 基本定义语法为:



函数名(有序参数表, 可变参数名1...; 键值参数表, 可变参数名2...) # 注意此处省略号有实际意义

其中, 可变参数名之后会有三点省略号, 标识其为可变参数。有序和无序部分都可以提供可变参数, 但均只能有一个且需要放在其他常规参数之后, 即紧邻界定符之前。

例如, 定义如下对任意多个参数进行打印的函数:

```
julia> function MyShow(x, y, z...)          # x与y是有序参数, z是可变参数
    print(x, " ", y, " ", z)
end
MyShow (generic function with 1 method)
```

以不同数量的参数对其调用时, 结果如下:

```
julia> MyShow(1, 2)
1 2 ()
julia> MyShow(1, 2, 3)
1 2 (3,)
julia> MyShow(1, 2, 3, 4)
1 2 (3, 4)
```

可见, 可变参变量会以元组结构记录未被参数表“取走”的剩余实参。如果无剩余实参, 该可变参变量会成为零元的元组。在函数内部, 再采用元组的访问方式解开可变参变量中的元素, 例如索引遍历或变量提取等, 便于对其实现各处理过程。

另外, 可变参数部分也可以进行类型限定, 例如:

```
julia> f(args::Int32...) = args          # 限定可变c参数args为Int32类型
f (generic function with 1 method)

julia> f(Int32(1))
(1,)

julia> f(Int32(1), Int32(2))
(1, 2)

julia> f(Int32(1), Int64(2))              # 可变参数2是Int64类型, 所以出错
ERROR: MethodError: no method matching f(::Int32, ::Int64)
Closest candidates are:
  f(::Int32...) at REPL[38]:1
```

可见, 其中的 `f()` 函数只支持类型均为 `Int32` 类型的任意参数。

基于可变参数机制, 我们便可以在调用时提供任意的参数, 在内部实现灵活的处理操作。例如, 一个支持任意数量的元素进行线性组合的操作函数:

```
julia> function linear_combine(args...)
    s = 0
    for p in args
        s += p[1]*p[2]
    end
    s
end
linear_combine (generic function with 1 method)
```

之后, 便可将系数与数值成对提供, 实现加权的线性组合:



```
julia> x = ((4,3+2im), (3.2,3//5));
```

```
julia> linear_combine(x...)
13.92 + 8.0im
```

或者

```
julia> x = ((2,1.1), (1,3), (4,3+2im), (3.2,3//5));
```

```
julia> linear_combine(x...)
19.119999999999997 + 8.0im
```

当然，存在可变参数的函数在调用时也可以采用可迭代数集展开的方式提供实参。但是因可变参数的存在，不再像上述（参见 7.1.4 节）那样有个数的限制，而是可以更多些。例如：

```
julia> d = (2, 3, 4);
```

```
julia> MyShow(1, d...)
1 2 (3, 4)
```

```
julia> MyShow((1,2,3,4)...)
1 2 (3, 4)
```

此时在所需实参被传递到参变量后，多余的元素会被“余留”在可变参量中，同样会传入到函数内，程序接收后可视需要进行适当的处理。

对于键值部分的可变参数，用法与上面所述的相差不大，但传入到函数内部时，不再是元组，而是命名元组。例如：

```
julia> varkws(;x=0, kwargs...) = (println(x); kwargs) # 键值部分只有一个键值对，键名为x
varkws (generic function with 1 method)
```

调用时，结果类似于：

```
julia> varkws(x=1, y=2, z=3) # y=2, z=3是额外，被处理为可变参数
1
pairs(::NamedTuple) with 2 entries:
 :y => 2
 :z => 3
```

```
julia> d = Dict{:x=>8.8, :y=>9.9, :z=>10.10}
Dict{Symbol,Float64} with 3 entries:
 :y => 9.9
 :z => 10.1
 :x => 8.8
```

```
julia> varkws(; d...)
8.8 # :x=>8.8 被提取到x参变量中
pairs(::NamedTuple) with 2 entries:
 :y => 9.9
 :z => 10.1
```

可见，未在参数表列出的键值对会被封装到可变参量中，并被记录在命名元组中；同样支持以字典数集的展开方式提供实参，且会自动传递到同名的参变量中。

如果可变参数需要严格限制个数，则使用 `Vararg{T,N}` 这种参数化类型对可变参数进行约束，其中，`T` 用于限制参数类型，`N` 则指定了必需的参数个数。例如：




```
julia> function varn(x, y, z::Vararg{Any,2}) # 限定额外仍需两个参数
    println(x)
    println(y)
    println(z)
end
varn (generic function with 1 method)
```

尝试调用一下：

```
julia> varn(1, 2, 3, 4)
1
2
(3, 4)
```

或者使用多元结构展开的方式：

```
julia> x = (1, 2, 3, 4)
(1, 2, 3, 4)

julia> varn(x...)
1
2
(3, 4)
```

但如果参数个数不满足要求，会报错，例如：

```
julia> varn(1, 2)
ERROR: MethodError: no method matching varn(::Int64, ::Int64)

julia> varn(1, 2, 3)
ERROR: MethodError: no method matching varn(::Int64, ::Int64, ::Int64)

julia> varn(1, 2, 3, 4, 5)
ERROR: MethodError: no method matching varn(::Int64, ::Int64, ::Int64, ::Int64, ::Int64)
```

看起来 Vararg 的作用有些鸡肋，但若 N 值是外部某个条件控制的，还是非常有用的。

7.3 函数对象

7.3.1 Function 类型

Julia 中的函数也是可操作的对象，也有自己的类型，即 Function，而且其类型也是 DataType，是 Julia 类型拓扑树的一部分；而定义的任何一个函数实现都是 Function 的实例。例如：

```
julia> isa(addtwo, Function)
true
```

此外，前文介绍的众多运算符在本质上也是函数对象，仅语法上有些特别而已，即：

```
julia> isa(+, Function) == isa(-, Function) == isa(*, Function) == isa(/, Function)
== true
true
```

而且这些运算符号完全可以作为函数名使用。例如：



```
julia> +(1, 2, 3)
6
```

```
julia> -(9, 2)
7
```

```
julia> *(2, 3, 4)
24
```

```
julia> /(6, 3)
2.0
```

当然，每个符号对参数个数等方面的约束会有所不同，使用这种方式时需要注意。

类似于其他的常规对象，函数对象也可以进行一些操作，包括“是否相等”或是赋值等操作。例如通过赋值操作便可得到源函数的别名函数：

```
julia> addt = addtwo
addtwo (generic function with 1 method) # 注意提示的函数名，并不是新的名字

julia> addt(1, 3)
4
```

对运算符同样适用，例如：

```
julia> add = +
+ (generic function with 163 methods) # 注意这里的163，我们会在介绍分发机制时说明此数字的意义

julia> add(1, 3, 4)
8
```

或通过是否“相等”或“相同”的运算符，判断它们是否是同一种实现，即：

```
julia> addt === addtwo
true

julia> notaddtwo == addtwo
false
```

由此可见，“在一切皆对象”的 Julia 中，函数可以有着更多灵活的用法。

另外，基于 Julia 对 Unicode 的强大支持，定义函数时完全可以使用 Unicode 字符命名函数，例如：

```
julia> Σ(x, y) = x + y
Σ (generic function with 1 method)

julia> Σ(2, 3)
5
```

在公式众多的代码中，如果采用这些具有数学意义的符号定义函数，能够让表达式看起来更为自然、直观。

7.3.2 函数作为参数

函数作为 Julia 中的对象，除了能够直接进行赋值等操作外，还可以作为其他函数的参数。例如，定义一个名为 `g` 的函数，其第一个参数是函数，代码如下：



142 ❖ Julia 语言程序设计

```
julia> function g(f, x, y)
    f(x, y)
end
g (generic function with 1 method)
```

不过此时最好能限定参数为 `Function` 类型，即：

```
julia> function g(f::Function, x, y)
    f(x, y)
end
```

此后，便可将已有的函数名直接作为参数传入进行调用，例如：

```
julia> g(addtwo, 1, 2)
3
```

运算符同样也可以作为函数参数，例如：

```
julia> g(+, 1, 2)
3
```

或者：

```
julia> g(-, 1, 2)
-1
```

```
julia> g(*, 1, 2)
2
```

```
julia> g(/, 1, 2)
0.5
```

不过函数实参采用的是“传值”方式，而不是引用传递；如果在内部被改变，不会影响外部原来的定义。例如：

```
julia> function h(f, x, y)
    f = -
    f(x, y)
end
h (generic function with 1 method)
```

```
julia> h(addtwo, 1, 2)
-1
```

```
julia> addtwo(1, 2)
3
```

其中，`addtwo` 传入 `h()` 后，被修改为减法运算符，但在外部调用 `addtwo()` 时仍是原来的加法。

显而易见，函数对象可以作为参数的默认值。例如：

```
julia> function h(x, y, f = addtwo)
    f(x, y)
end
h (generic function with 2 methods)
```

```
julia> h(1, 2)
```

默认使用 `addtwo()` 函数对象



3

```
julia> h(1, 2, -)          # 调用加法运算符
-1
```

其中 `h()` 将其函数参数 `f` 默认为 `addtwo()` 函数，当未提供时会被隐式地调用，执行 `x` 与 `y` 的加法操作；提供了特定的函数（运算符）- 之后，内部便以实际提供的方法进行处理。



提示 函数对象在作为参数时并没有特别的限制，在开发中灵活地应用，能够让程序具备所需的灵活性与延展性。

7.3.3 函数作为返回值

同样，函数可以作为结果，作为其他函数的返回值。例如：

```
julia> function get_addtwo_func()
    addtwo          # 将函数addtwo返回
end
get_addtwo_func (generic function with 1 method)

julia> get_addtwo_func()          # 调用该函数，结果返回函数对象
addtwo (generic function with 1 method)
```

其中，`get_addtwo_func()` 在被调用时，返回的结果是“generic function”即函数对象，而不再是某个计算值。当然，运算符本身也可以作为结果返回，例如：

```
julia> function g()
    +                # 将运算符+返回
end
g (generic function with 1 methods)

julia> g()          # 调用该函数，结果返回函数对象
+ (generic function with 163 methods)
```

虽然看起来有些奇怪，但这正是 Julia 独具特色的地方：一切皆对象。其中的加法符号便是一个函数对象，能够像其他常规对象那样进行各种支持的操作。

如果要直接对返回的函数对象进行调用，方法并不复杂，只需在父函数调用形式后再附加返回函数的参数表即可。例如：

```
julia> f()(1, 2)          # 调用addtwo函数
3

julia> g()(1, 2)          # 调用+运算符
3
```

若是觉得这样不方便，可以先将返回的函数对象绑定到一个变量上，再进行调用，例如：

```
julia> at = f()
addtwo (generic function with 1 method)

julia> at(1, 2)
3
```



实际上，函数在定义时便会生成函数对象。可见，函数内部是可以再次定义一个函数的，例如：

```
julia> function getfunc()
    function addthree(x, y, z)
        x + y + z
    end
end
getfunc (generic function with 1 method)
```

其中，在 `getfunc()` 内部又定义了 `addthree()` 函数，用于累加三个变量的值。调用后便可发现，返回的同样是“generic function”对象：

```
julia> add3 = getfunc()
(::addthree) (generic function with 1 method)
```

而且可以正常调用，例如：

```
julia> add3(1, 2, 3)
6
```

其中，`add3()` 对传入的 3 个实参实现了累加操作。

对于一些简单的、通用性不强的特定函数，可以在内部单独定义，避免与外部的函数冲突。也能够一定程度上达到逻辑集中的目的，可以成为 Julia 特有的编程范式。但是使用 `function` 关键字略显臃肿，下面将介绍更为简洁的方式。

7.4 匿名函数

在函数作为参数或返回值时，如果定义体较为简单，可以不用预先定义该函数，而采用更为简单、直接的匿名函数。匿名函数不需要函数名，只须提供参数需求及实现体即可。其定义的基本语法为：

```
function (参数表)
    # 实现体
end
```

或者采用另外一种更为常用的方式，即：

```
(参数表) -> 复合表达式
```

其中，符号 `->` 用于连接函数参数与实现体部分。在后一种简单的方式中，如果参数只有一个，圆括号可以省略；但无参数或多于一个参数时，圆括号不可省略。

下面是一些匿名函数的例子：

```
julia> (x, y, z) -> x * y + z
(::#1) (generic function with 1 method)

julia> x -> 2x                                # 只有一个参数，不用圆括号
(::#3) (generic function with 1 method)

julia> () -> 10 + 2                            # 无参数，圆括号不能省略
(::#5) (generic function with 1 method)
```



从成功信息看，与前面介绍的普通非匿名函数相同，都是“generic function”的一种。而且在定义后，Julia 内部会自动给予该匿名函数一个编号。

不过需要留意匿名函数与常规表达式的区别，它们在本质上是不同的。表达式是一个可执行的操作，而匿名函数则是建立了参数表到表达式的映射。

在需要函数作为参数时，如果是简短的实现便无须先定义再使用，而是在调用处直接提供匿名函数。例如：

```
julia> function g(f, x, y, z)
    f(x, y, z)
end
g (generic function with 1 methods)

julia> g( (x, y, z) -> x * y + z, 2, 3, 4) # 首个参数为函数
10                                     # 即 2*3+4

julia> function h(f::Function)
    f()
end
h (generic function with 2 methods)

julia> h( () -> 10 + 2 )                # 无参的匿名函数，直接返回两个值的累加和
12
```

此外，匿名函数作为返回值也极为简单，只需将其作为函数体内部的最后一个表达式即可，或使用 return 返回也可以。例如：

```
julia> function f1()
    x -> x^2 + 2x - 1
end
f1 (generic function with 1 method)

julia> function f2()
    return (x, y) -> x + y
end
f2 (generic function with 1 methods)

julia> f1()(2)
7

julia> f2()(2, 3)
5
```

事实上，匿名函数除了没有名字之外，其他的语法均遵循普通函数的定义规则，例如可进行类型限定，使用默认值、可变参数或键值参数等，这里不再赘述。

但较长的匿名函数直接以实现体作为参数时，会使代码变得极难阅读，也不利于维护。为解决该问题，Julia 提供了 do 代码块的语法，能够在调用时以独立的结构实现匿名函数，基本表述方式为：

```
调用函数名(实参表) do 匿名函数参变量表
    # 匿名函数实现体
end
```



其中“调用函数名”是指已经存在的某个函数名，且该函数原型中的首个参数必须能够接收函数 Function 类型；实参表则是除了首个函数参数外的其他参数值；而“匿名函数参变量表”则是待定义的匿名函数各参数，以逗号分隔。

以之前的函数 `g(f, x, y, z)` 为例，可通过 `do` 代码块传入匿名参数对其实现调用，例如：

```
julia> g(2, 3, 4) do x, y, z
           x + y + z
       end
9
```

其中，`do x, y, z` 代码块会创建一个含有三个参数的匿名函数，并将其传递给函数 `g()` 作为第一个实参；而 `g()` 函数的其余参数则通过自己的实参表提供。

另外，无参的匿名函数同样也可以使用 `do` 代码块，以上述的 `h()` 函数为例：

```
julia> h() do
           10 + 2
       end
12

julia> h() do
           println("nothing to do")
       end
nothing to do
```

此时，因为匿名函数不需要参数，所以 `do` 之后无须提供任何参变量。

若是函数的实现很简短，`do` 代码结构也可以写在同一行，例如：

```
julia> g(10, 20, 30) do x, y, z x^2+x*y-z end
270
```

或者：

```
julia> g(10, 20, 30) do x, y, z a = x^2+x*y-z; b=0.5*a end
135.0
```



多个实现语句时，最好用分号分开。

7.5 参数化方法

为了避免因不受控制的类型而引发难以预料的结果，在开发时应尽可能地限定参变量的类型。但是这种限定仍是有局限的，例如，无法限定某些参数必须是同类型。在前述类型限定的时候，除非类型是元类型，否则即便限定的抽象类型一致也无法保证不同参数一定有着相同的具体类型。例如，`addtwo(x::Integer, y::Integer)` 中的 `x` 可以是 `UInt8` 类型，而同时 `y` 则可以是 `Int32` 类型。

在 Julia 中，不但抽象类型、元类型或复合类型可以参数化，作为独立类型的函数也是可以参数化的。基本方式是在函数定义时以类型参数对变量的类型进行限定，形式为：

```
function 函数名(参数1::T1, 参数2::T2, ...) where {类参下界 <: T1 <: 类参上界, T2, ...}
    # 实现
end
```

或者:

```
函数名(参数1::T1, 参数2::T2, ...) where {类参下界 <: T1 <: 类参上界, T2, ...} = 复合表达式
```

其中, 在函数参数表后采用了 where 关键字的表达方式, 并逐一给出类参的限界要求(仅以首个类型 T1 作为示例, 其他的类同)。

显然参数化作为对参数类型的另外一种控制方法, 比常规的类型限定更为灵活, 不但可以限定类型的父类型, 也可以更为直观地约束参数间的类型相关性。例如:

```
f1(x::T1, y::T2) where {T1 <: Real, T2 <: Integer}
f2(x::T, y::T) where {T}
```

其中, f1() 函数中限定参数 x 为 Real 的任意子类型, y 只能为整型 Integer 的子类型; 而函数 f2() 中则不限定 T 为何种类型, 但要求参数 x 与 y 必须类型相同。再例如:

```
julia> same_type(x::T, y::T) where {T} = true
same_type (generic function with 1 method)
```

使用相同类型的参数进行调用, 能够成功:

```
julia> same_type(Float32(1.1), Float32(2.2))
true
```

```
julia> same_type((1,2), (3,4))
true
```

但如果类型不同, 则会报错:

```
julia> same_type(2.0, 3)
ERROR: MethodError: no method matching same_type(::Float64, ::Int64)
Closest candidates are:
  same_type(::T, ::T) where T at REPL[24]:1
```

即便以相同的抽象类型限定也不行:

```
julia> same_type(Real(2.0), Real(3))
ERROR: MethodError: no method matching same_type(::Float64, ::Int64)
Closest candidates are:
  same_type(::T, ::T) where T at REPL[24]:1
```

因为实参值本身只能是具体类型, 抽象类型并不能真正实现对实参的类型约束; 而且 Julia 内部不会对参变量进行类型提升或转换。所以, 虽然在 same_type() 函数的定义中, 没有对类参的类型作限界处理, 理论上两个参数可以是 Any 类型, 但它们的具体类型必须是相同的。

如果为 same_type() 函数增加一个方法(下文介绍):

```
julia> same_type(x::T1, y::T2) where {T1 <: AbstractFloat, T2 <: Integer} = true
same_type (generic function with 2 methods)
```

此时, same_type() 函数拥有了两个方法。再次重试调用该函数, 便可顺利通过:

```
julia> same_type(2.0, 3)
```



```
true
```

当然，如果类型不满足其中父类型的约束，仍会出错：

```
julia> same_type(2.0, 3//5)
ERROR: MethodError: no method matching same_type(::Float64, ::Rational{Int64})
Closest candidates are:
  same_type(::T1<:AbstractFloat, ::T2<:Integer) where {T1<:AbstractFloat,
    T2<:Integer} at REPL[30]:1
  same_type(::T, ::T) where T at REPL[24]:1
```

这种函数参数化技术中，因为类型被参数化，所以在函数中能够对类型进行直接操作。

例如，类型本身可直接作为返回值：

```
julia> mytypeof(x::T) where {T} = T
mytypeof (generic function with 1 method)

julia> mytypeof(1)
Int64

julia> mytypeof(1.0)
Float64
```

或者对参数的类型做一致性断言：

```
julia> is_same_type(x::T1,y::T2) where {T1,T2} = T1==T2
is_same_type (generic function with 1 method)

julia> is_same_type(1, 1.0)
false

julia> is_same_type(3//5, 8//9)
true
```

这种方式中，当 where 后的“类型列表”中只有一个时，大括号是可以省略的，不过为了代码清晰，最好保留。另外，类型列表有多个时，也可以使用多个 where 分开表示。例如：

```
julia> f(x::T1, y::T2) where T1 <: Number where T2 = true
f (generic function with 2 methods)
```

虽然 Julia 是弱类型语言，但仍然提供了包括类型限定、参数化等多种类型控制的方法，这不仅仅是对约束性逻辑实现的支持，同时也是性能的需要。在类型明确的情况下，不仅可尽早地发现程序存在的问题，也能够帮助编译器减少运行期的负载，提升编译与执行效率。

7.6 多态分发

从数学意义上讲，函数的本质是一种建立了参数组合与结果值之间映射关系的对象。例如，四则运算中的加法，可以针对不同的数值类型，也可以针对不同个数的操作数。但在程序实现中，因为具体类型的限制，落实这种概念往往需要针对各种参数个数及类型的

组合定义出多个函数。虽然我们能够通过抽象类型的限定方式或参数化方法达到一定的目的，但仍然无法满足多样性的需求。

在实践中，我们常常希望同映射概念的函数能够以统一的方式实施，而且在不断迭代的过程中，该概念能够在实现中不断延展扩充，以适应业务逻辑等方面的变化。一个基本的方法是，同一概念能够使用同一函数名进行定义，无须为不同的参数组合给定不同的函数名以作区分，而在调用时能够自动根据提供的实参信息自动找到同名函数的对应实现。

如前文所述，当在 REPL 中成功定义一个函数时，会出现如下的提示：

```
函数名 (generic function with 1 method)
```

但如果使用同样的函数名再定义另外一个参数类型不同的实现，提示会略有不同：

```
函数名 (generic function with 2 methods)
```

其中 method 数量发生了变化。例如：

```
julia> f(x::Int32) = Int32
f (generic function with 1 method)
```

```
julia> f(x::Int64) = Int64
f (generic function with 2 methods)
```

其中，两个 `f()` 函数同名，区别仅在于唯一的参数采用了不同的类型限定；而上报的成功信息表示：作为“generic function”的 `f()` 具备了两个“methods”。

在 Julia 中，将某个同名函数不同的定义称为该函数的实现方法（Methods），而且会在内部对这些方法进行跟踪，并记录在内部的方法列表中。

如果我们尝试调用上例中的 `f()` 函数，会发现：

```
julia> f(Int32(1))
Int32
```

```
julia> f(1)
Int64
```

显然，两次调用的都是 `f()` 同名函数，但实际执行的却是不同的实现。实际上，在对 `f()` 调用时，Julia 能够根据实参情况自动在方法表中找到匹配的实现。

这种在同名函数的方法列表中寻找、匹配对应实现或定义的调用过程被 Julia 称为多态分发（Multiple Dispatch）。这种分发过程会根据函数名获得其方法列表，然后根据其中的参数个数与类型约束，选择最为匹配、恰当的实现方法，之后才会传参、启动、执行该方法。

对于任意的函数对象，如果查询其有几种方法，直接在 REPL 输入其名称即可。例如：

```
julia> f
f (generic function with 2 methods)
```

即 `f()` 函数有 2 个实现方法。另外，也可以使用 `methods()` 函数查看更为详细的信息，包括每个方法的原型，甚至定义的代码位置，例如：

```
julia> methods(f)
# 2 methods for generic function "f":
[1] f(x::Int64) in Main at REPL[2]:2
[2] f(x::Int32) in Main at REPL[1]:2
```

可以说，多态分发在抽象类型限定及参数化的基础上，从另外一个角度提供了对既有功能的延展机制，能够在各方法约束尽可能严格的情况下，通过多次定义同名实现体达到多种类型兼容的目的，而且还能够在参数个数迥异的情况下实现概念的统一。所以，多态分发给 Julia 中的应用极为广泛。例如，内置的加法运算符 + 的实现方法约有 163 个之多，例如：

```
julia> methods(+)
# 163 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:277
[2] +(x::Bool, y::Bool) in Base at bool.jl:104
[3] +(x::Bool) in Base at bool.jl:101
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:112
[5] +(x::Bool, z::Complex) in Base at complex.jl:284
[6] +(a::Float16, b::Float16) in Base at float.jl:392
[7] +(x::Float32, y::Float32) in Base at float.jl:394
[8] +(x::Float64, y::Float64) in Base at float.jl:395
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:278
[10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:292
... # 已省略
```

可见，其支持的参数组合方式非常多，包括元类型、抽象类型及参数化的定义，等等。

但需要注意的是，定义同名方法时，如果参数组合结构（个数与类型）完全一致，会覆盖之前存在的方法，并不会新增方法。例如：

```
julia> g(x::Int64, y::Int64) = 1
g (generic function with 1 method)

julia> g(x::Int64, y::Int64) = 2
g (generic function with 1 method)
```

连续两次定义的 `g()` 并没有增加新方法，此时仅最后一个方法起效。即：

```
julia> g(1,2)
2
```

调用时返回的是 2 而不是 1，因为返回 1 的定义方法被后一个覆盖了。而且这种覆盖情况还会发生在有默认参数的时候。例如，针对以下的复合结构：

```
mutable struct Point2D{T}
    x::T
    y::T
end
```

首先定义一个移动函数，修改其内部坐标值的位置：

```
julia> function movepoint(p::Point2D, ox = 1, oy = 2)    # ox和oy为控制偏移的参数
    p.x += ox
    p.y += oy
    p
end
movepoint (generic function with 3 methods)
```


我们会发现，虽然该 `movepoint()` 被首次定义，但成功信息中提示有 3 个方法。查看如下：

```
julia> methods(movepoint)
# 3 methods for generic function "movepoint":
[1] movepoint(p::Point2D) in Main at REPL[23]:2
[2] movepoint(p::Point2D, ox) in Main at REPL[23]:2
[3] movepoint(p::Point2D, ox, oy) in Main at REPL[23]:2
```

原因在于，带有默认参数的函数调用会存在不同个数的调用结构，为了区分，Julia 内部会预先将这些调用形式自动展开为对应的方法列表，以便能够快速、准确地找到最佳的方法。

所以如果我们再定义只需一个偏移参数的同名方法，会发现方法列表数量没有增加：

```
julia> function movepoint(p::Point2D, offset)
    p.x *= offset
    p.y *= offset
    p
end
movepoint (generic function with 3 methods)
```

而且其中一个原来被自动展开的方法被替换掉了，如下所示：

```
julia> methods(movepoint)
# 3 methods for generic function "movepoint":
[1] movepoint(p::Point2D) in Main at REPL[23]:2
[2] movepoint(p::Point2D, offset) in Main at REPL[25]:2 # 替换到了原来的movepoint(p::
    Point2D, ox)
[3] movepoint(p::Point2D, ox, oy) in Main at REPL[23]:2
```

尝试以两个参数调用该函数，如下：

```
julia> movepoint(p, 10)
Point2D{Int64}(120, 130)
```

可见，执行的是 `offset` 对应的方法，将成员值都放大了 10 倍，而不是累加 10。

另外一个在使用分发机制时需要特别注意的是方法歧义 (Method Ambiguities)。因为默认参数、可变参数以及抽象类型限定 (若无限定实际相当于限定为 Any 类型) 等方式的存在，经常会导致参数表在类型兼容方面出现重叠现象。

例如，分别定义 `h()` 函数的两个方法，如下所示：

```
julia> h(x::Float64, y) = 2x/y
h (generic function with 1 method)
```

```
julia> h(x, y::Float64) = x + 2y
h (generic function with 2 methods)
```

这两个成功创建的方法在形式上仅是类型限定不同，且都会存在于 `h()` 的方法列表中：

```
julia> methods(h)
# 2 methods for generic function "h":
[1] h(x::Float64, y) in Main at REPL[11]:1
[2] h(x, y::Float64) in Main at REPL[12]:1
```

如下的调用是没有问题的：


```
julia> h(1.1, Int64(3))           # 执行第一个
0.7333333333333333

julia> h(Int64(3), 1.1)           # 执行第二个
5.2
```

两次调用时, x 与 y 的类型都互不相同, 而且只满足两个方法中的一个。但如果提供的两个实参都是 `Float64` 类型, 调用会失败:

```
julia> h(2.0, 3.0)
ERROR: MethodError: h(::Float64, ::Float64) is ambiguous. Candidates:
  h(x::Float64, y) in Main at REPL[11]:1
  h(x, y::Float64) in Main at REPL[12]:1
Possible fix, define
  h(::Float64, ::Float64)
```

因为其符合所有的两个方法原型, 所以 Julia 的分发机制无法判断使用哪个, 只能上报歧义性错误, 但也同时给出了修正该建议的策略, 即需要定义一个准确符合该次调用的、类型约束更严格的实现方法, 即:

```
julia> h(x::Float64, y::Float64) = 2x+2y
h (generic function with 3 methods)    # 新增了一个方法

julia> h(2.0, 3.0)
10.0
```

可见 Julia 对调用执行多态分发时, 会优先匹配条件最严格的实现, 其次才是较宽松的方法。

再看一个存在可变参数的例子, 例如:

```
julia> y(a,b) = -a-b
y (generic function with 1 method)

julia> y(args...) = +(args...)
y (generic function with 2 methods)

julia> methods(y)
# 2 methods for generic function "y":
[1] y(a, b) in Main at REPL[1]:1
[2] y(args...) in Main at REPL[2]:1
```

其中, `y()` 函数定义了两个方法, 前一个只接收两个参数, 另一个则接收任意数量的参数, 这期间没有重叠覆盖的情况出现。尝试调用如下:

```
julia> y(1)           # 执行了y(args...)
1

julia> y(1, 2)         # 执行了y(a,b)
-3

julia> y(1, 2, 3)      # 执行了y(args...)
6
```

显然, 其中的调用 `y(1, 2)` 符合 `y(args...)` 原型, 但选择的是 `y(a, b)` 实现, 因为其参数的要求更为明确。所以, 虽然存在多种选择, 但 Julia 内部的优选机制能够自动解决

这样的冲突，不会报错。

显而易见，一个函数的实现方法越多，发生歧义的可能性就越大。为此，官方文档也给出了不少可操作的建议，有兴趣的读者可以查阅相关资料。

Julia 的弱类型机制能让开发更快捷，但恰当的类型约束能够为程序高效、正确地执行提供保障。我们需要在具体化和抽象化之间选择合适的平衡点，既要避免过度的类型限定带来代码的膨胀及维护成本，也要确保每个函数都能够按照预期的方式准确执行，且不会出现难以预料的问题。而寻找这种平衡点的关键是，如何更好地利用 Julia 的多态分发机制。

为此我们需要更多的实践，只有熟悉了 Julia 语言的各种语法规则及特点，才能够让 Julia 程序同时具备正确性、兼容性、延展性及灵活性。

7.7 复合类型构造方法

正因为 Julia 独特的多态分发机制，所以官方建议在复合类型的外部实现自定义的构造方法。也正因为此，我们将这部分内容放在本章进行介绍。当然，Julia 并没有限制不能在复合类型内部创建构造方法，只是有些特别而已。

7.7.1 外部构造方法

显而易见，复合类型的构造函数本质上也是一种函数，是 `Function` 的实例之一，所以其定义方式也遵循上文所介绍的函数有关的语法规则，能够定义出各种不同的构造方法。但是，构造函数与常规函数也有着很大的区别：一是构造函数的名称应约定为复合类型名称（即两者名称一致），二是新增的方法均要以默认或已有的构造方法为基础才能创建出复合类型的实例对象。

定义在复合类型结构内部的被称为内部构造方法，而外部定义的则被称为外部构造方法。一般而言，只要在复合类型声明的作用域内，外部构造方法可以定义在代码的任意位置。

例如，前面曾声明的复合类型 FooA，如下所示：

```
struct FooA
  a
  b::Float64
end
```

其存在两个默认构造方法:

```
FooA(a, b::Float64)
FooA(a, b)
```

如果希望能够通过一个参数构造出两个成员均相同的 FooA 对象，则可以在其外部增加一个自定义的构造方法，如下：

[illegible]

此后便可调用：

```
julia> FooA(1.0)
FooA(1.0, 1.0)
```

或者，采用参数化函数的方式定义如下：

```
julia> FooA(x::T) where T<:Integer = FooA(2x, 3x+10.8)
FooA

julia> FooA(1)
FooA(2, 13.8)
```

此时，使用 `methods()` 查看该复合类型的所有构造方法原型，如下：

```
julia> methods(FooA)
# 4 methods for generic function "(:Type)":
[1] FooA(x::Float64) in Main at REPL[2]:1      # 新增的Float64参数的外部构造方法
[2] FooA(x::T) where T<:Integer in Main at REPL[5]:1 # 新增的外部参数化构造方法
[3] FooA(a, b::Float64) in Main at REPL[1]:2    # 有类型约束的默认构造方法
[4] FooA(a, b) in Main at REPL[1]:2            # 无类型约束的默认构造方法
```

可见，新增的外部构造方法也同样会进入复合类型的构造方法列表。

同样，在自定义构造方法时，我们可以为构造方法提供默认实参值，例如：

```
julia> FooA(m=1.0, n=2.0) = FooA(m, n)
FooA
```

此后，便可进行无参调用并创建 `FooA` 对象，即：

```
julia> FooA()
FooA(1.0, 2.0)
```

但需注意此时 `FooA` 构造方法列表的变化，如下：

```
julia> methods(FooA)
# 6 methods for generic function "(:Type)":
[1] FooA() in Main at REPL[9]:1      # 新增的构造方法，自动展开所得
[2] FooA(x::Float64) in Main at REPL[2]:1 # 原有保留的
[3] FooA(x::T) where T<:Integer in Main at REPL[5]:1 # 原有保留的
[4] FooA(m) in Main at REPL[9]:1      # 新增的构造方法，自动展开所得
[5] FooA(a, b::Float64) in Main at REPL[1]:2 # 原有保留的
[6] FooA(m, n) in Main at REPL[9]:1    # 新增的构造方法，但替换掉了原有的
# FooA(a, b)
```

其中，`FooA(m=1.0, n=2.0)` 本应自动生成 3 个新方法，使得总数增加至 8 个，但却只有 7 个，原因是其生成的两参数版将原本默认的构造方法覆盖了。所以此时若以两个参数调用构造方法，会调用原型为 `FooA(m=1.0, n=2.0)` 这个带默认值的版本。例如：

```
julia> FooA(2.0, 3.0)
FooA(2.0, 3.0)
```

需要明确的是，此时被调用的 `FooA(m, n)` 在创建对象时，实际执行的是 `FooA(a, b::Float64)` 原型。

可如果此时提供的参数是整型，则会报错，例如：

```
julia> FooA(3, 2)
ERROR: StackOverflowError:
```

不仅构建失败，而且导致了栈溢出异常。

原因在哪里？

实际上，`FooA(m=1.0, n=2.0) = FooA(m, n)` 在被调用后，其中的 `FooA(m, n)` 执行时也需要从构造方法列表中搜索匹配的原型；对于两个参数的调用，只有两个原型符合，即 `FooA(a, b::Float64)` 和 `FooA(m, n)`。但因为类型限定，`FooA(m, n)` 成了唯一的选择。但恰巧的是，该原型正是新增的外部构造方法 `FooA(m=1.0, n=2.0)` 自身，所以该构造方法对原默认方法的覆盖行为引发了循环递归调用的后果，并且无法终止，导致了栈溢出错误。这种“死亡递归”是需要开发者非常注意的隐蔽问题。

为此，在外部定义构造方法时，应避免覆盖默认方法。在创建对象或进行调用时，应尽量基于现有的尤其是默认的构造方法，尽量少地在外部新增。新增时，最好先查看原有的方法列表，避免引入冲突导致不可预料的后果。

另外，构造方法也可以采用可变参数。例如：

```
julia> function FooA(p, q...)
    FooA(p, +(q...))
end
FooA

julia> FooA(1.0, 2.0, 3.0, 4.0, 5.0)    # 注意不能再使用整型，否则溢出
FooA(1.0, 14.0)
```

同常规函数一样，可变参数版本的优先级也较低。

当然，构造方法未必一定要创建复合类型对象，也可任意返回其他类型的结果。例如：

```
julia> FooA(x, y, z) = (x, y, z)
FooA

julia> FooA(1, 2, 3)
(1, 2, 3)
```

事实上，此时的构造方法虽然与复合类型同名，但已经退化为普通函数。可见，无须通过同名的构造方法创建复合类型的实例对象，在常规函数中调用已有的构造方法即可；但此时的函数不再是构造方法的扩展延伸，只不过是普通的实例化后再使用而已。

7.7.2 内部构造方法

在复合类型内部自定义构造方法是 Julia 提供的另外一种选择。对此，同样有一些约束：一是内部出现的任意函数或方法都会“消灭”复合类型所有的默认构造方法，无论该函数是否与该复合类型同名；二是内部的构造方法中只能使用 `new` 关键字创建对象。

例如，有如下的复合类型：

```
mutable struct BarA
    a
end
```

此时其构造方法列表为：


```
julia> methods(BarA)
# 1 methods for generic function "(:Type)":
[1] BarA(a) in Main at REPL[41]:2      # 唯一的默认构造方法
```

随意在其内部定义一个函数，例如：

```
julia> mutable struct BarA
    a
    some(x) = 1                        # 随意定义的函数
end
```

```
julia> methods(BarA)
# 0 methods for generic function "(:Type)":      # 原来的构造方法消失了
```

可见，其中的默认构造方法 `BarA(a)` 消失了，导致此后再无法对该复合类型进行构造。而且这种情况与复合类型是否为 `mutable` 无关。

由此可见，Julia 并不像其他面向对象语言那样支持对成员函数的封装，复合类型内部只能放入一种函数，即内部构造方法。例如：

```
struct OrderedPair
    x::Real
    y::Real
    OrderedPair(a) = new(a, a)
end
```

此时的构造方法列表为：

```
julia> methods(OrderedPair)
# 1 methods for generic function "(:Type)":
[1] OrderedPair(a) in Main at REPL[2]:4      # 内部自定义的构造方法
```

之后便可基于其中定义的单参数方法创建实例，例如：

```
julia> OrderedPair(1)
OrderedPair(1, 1)
```

但有一种情况是需要特别注意的，如果重复声明复合类型（仅限名称、字段名及类型限定均一致，否则不构成重复声明并报 `redefinition` 错误），不同时刻出现的各自构造方法均会被加入到构造方法列表中。例如：

```
julia> struct OrderedPair
    x::Real
    y::Real
    OrderedPair(a, b) = new(b, a)      # 第二次重复声明
end                                     # 参数顺序故意地颠倒
```

```
julia> methods(OrderedPair)
# 2 methods for generic function "(:Type)":
[1] OrderedPair(a) in Main at REPL[7]:4      # 前一次中出现的构造方法
[2] OrderedPair(a, b) in Main at REPL[6]:4   # 第二次出现的构造方法
```

这也是对现有复合结构构造方法的一种扩充。

另外，在构造时，我们还可以对构造过程进行更好的控制。例如，参数不满足条件时不允许创建对象，并上报相应的错误：

```
julia> struct OrderedPair
    x::Real
    y::Real
    OrderedPair(x::Real, y::Real) = x > y ? error("out of order") : new(x, y)
end
```

此时可选的构造方法有：

```
OrderedPair(a, b)           # 两个参数，均Any类型
OrderedPair(a)              # 也是两个参数，但限制更严格
OrderedPair(x::Real, y::Real)
```

由于参数表 $(x::\text{Real}, y::\text{Real})$ 比 (a, b) 有更为具体的类型限制，所以当参数为实数类型时会优先调用 `OrderedPair(x::Real, y::Real)` 原型。例如：

```
julia> OrderedPair(1,2)
OrderedPair{Int64}(1, 2)

julia> OrderedPair(2,1)
ERROR: out of order          # 上报错误
```

可见，当第一个参数大于第二个参数时，上报了指定的错误。但若提供的是非实数实参，因为执行的原型为 `OrderedPair(a, b)` 这个构造方法，所以不会主动报错，例如：

```
julia> OrderedPair(2+0im, 1+0im)
OrderedPair{Complex{Float64}}(1, 2)
```

虽然提供的参数从数学意义上等效于实数，但毕竟是复数类型，所以执行的仍是另外一个构造方法，也不会报给定的错误。

自定义的内部构造方法同样也会成为其他外部构造方法的基础。在 Julia 开发中，可以利用内部构造方法这种特性，对内部可用的构造方法进行裁剪，实现对实例创建过程的精细控制，确保对象的变化在预期的范围内。

不过 Julia 设计者建议，在声明复合类型时至少能给出一个为全部成员提供参数的内部构造方法，并能够配有强制的错误检查。其他为了便利而提供的构造方法，例如带有默认参数的方法、复合类型的辅助转换等，最好以外部构造方法的形式提供；这样基于已知的、可预期的内部构造方法，可更安全地提升与增强构造函数的功能。

多维数组

数组在数学表达中是非常常见的，在科学计算的各种开发场景中也是最为基础的数据结构。但在 C++、Java 等常见的语言中，不提供矩阵或高维数组的直接支持，在需要使用高维结构时，数据方面的操作会非常麻烦，尤其是涉及动态大小时，使用起来也非常不方便，尤其在像 C++ 这种语言中，复杂嵌套的内存管理会带来很大的工作量，而且容易出错。

Julia 则设计了原生的多维数组结构，支持任意维度的数组，同时在数组结构上提供了大量便利的操作支持，而且开发者还能够在其基础上进行各种扩展。仅此一点，就为 Julia 建立了极大的发展优势。

8.1 创建数组

在 Julia 中可以定义任意维度的数组。常见的数组有两种结构：一维与二维。其中一维数组又称为向量，分为行向量与列向量两种，在 Julia 中默认为列向量；二维数组即矩阵，第一维是行，第二维是列。无论哪一种结构，每一维的大小与长度均可以在定义时指定，也可以定义后动态调整。

为了方便后面描述，特此约定两个用词：

- 维度：一维指向量，二维指矩阵，以此类推。
- 阶数：每一个维度上元素的个数，即各维的长度。

例如， N 阶列向量，指的是一个列向量长度为 N ，即有 N 个元素；二维数组的阶为 $M \times N$ ，则是指有 M 行 N 列，即每列中 M 个元素，长度为 M ，而每行有 N 个元素，长度为 N 。

在 Julia 中,任何数组类型都是参数化抽象类型 `AbstractArray{T, N}` 的子类型。其中 `N` 是数组的维数,而 `T` 是元素的类型,几乎支持任意的类型,包括自定义的复合类型。

对于所有继承自 `AbstractArray` 的类型,都可通过以下几个函数取得数组的属性信息,如表 8-1 所示。

表 8-1 数组属性函数

函 数	描 述
<code>eltype(A)</code>	数组 A 中的元素类型
<code>length(A)</code>	数组 A 中的元素个数
<code>ndims(A)</code>	数组 A 的维数
<code>size(A)</code>	数组 A 各维度的阶数,返回 tuple 结构
<code>size(A,n)</code>	数组 A 第 n 维的阶数

在所有的 `AbstractArray` 子类型中,最为常用的是 `Array` 子类型,其继承路径为:

```
Array{T,N} <: DenseArray{T,N} <: AbstractArray{T,N}
```

其中, `DenseArray` 表示一类“稠密”数组,指的是内部元素均以直接、简朴的方式记录在结构中,都在内存中占有“自己”的位置,有着实际的内存实例。与之不同的是,前文介绍过的 `Range` 类型仅在内部结构中表达了元素的生成方式,并没有实际地存储元素,自然也不是稠密类数组,即不是 `DenseArray` 的子类型。

创建数组的基本方式是通过类型的构造方法:

```
Array{T, N}(undef, dims)
```

其中, `N` 是维度大小; `T` 限定了元素类型; 参数 `dims` 用于指定各维阶数,可以是元组或一系列整型值,但元组长度或阶数实参的个数需与 `N` 一致。为简便,可省略维度参数,而采用 `Array{T}(undef, dims)` 形式。参数中出现的 `undef` 等效于 `UndefInitializer()` 函数,即:

```
julia> undef === UndefInitializer()
true
```

类型 `UndefInitializer` 是 v1.0 版新增的内容,是一个说明数组处于尚未初始化状态的单例类型,而 `undef` 只是其无参数构造方法的别称。

另外,为了使用的方便,对于一维数组,可使用 `Vector` 类型指代,即向量;而二维数组,可使用 `Matrix` 类型指代,即:

```
julia> Vector
Array{T,1} where T
```

```
julia> Matrix
Array{T,2} where T
```

可见 `Vector` 和 `Matrix` 是 `Array` 在不同维度时的类型别称。

下面看一些采用数组构造方法创建数组对象的例子:


```

julia> A = Array{Float64, 2}(undef, 2, 3)
2×3 Array{Float64,2}:
 8.88324e-316  6.17328e-316  6.17146e-316
 3.16865e-315  3.16865e-315  0.0

julia> B = Array{Int32}(undef, 2,3)          # 省略了维度类参
2×3 Array{Int32,2}:
 614238096  614238192  614238256
      0      0      0

julia> Vector(undef, 3)
3-element Array{Any,1}:
 #undef
 #undef
 #undef

julia> Matrix{Int}(undef, (2, 4))
2×4 Array{Int64,2}:
 1   8  13  17
 3  11  14  27

```

需要注意的是,正如 `undef` 显式说明的那样,构造方法创建的数组都是未初始化的,所以其中元素的初始值都是随机数值,使用前最好要按需进行恰当的初始化。当然,除了构造方法外,Julia 中还提供其他更为方便的数组创建方式,下面逐步介绍。

8.1.1 串联方式

事实上,创建数组有一种最直接、最简单的方式,便是在方括号 `[]` 中直接罗列出所有的元素值。例如:

```

julia> [1.2 2.3 3.4 5 6]
1×5 Array{Float64,2}:
 1.2  2.3  3.4  5.0  6.0

julia> [1.2 'a' "abc" Some(3)]
1×4 Array{Any,2}:
 1.2  'a'  "abc"  3

```

在这种罗列方式中,Julia 会自动根据其中所有元素的类型,进行必要的类型转化与提升,然后以统一的类型创建 `Array` 对象。不过上例中使用空格分离各个元素(空格的数量不会影响效果)创建的是行向量,即第一维的阶数为 1 的二维矩阵。或者说,以空格字符罗列出的数值或变量均作为同一行的元素处理。

如果要创建列向量,则需要使用逗号或分号来隔离各个元素,例如:

```

julia> [1.2; 2.3; 3.4; 5; 6]
5-element Array{Float64,1}:
 1.2
 2.3
 3.4
 5.0
 6.0

julia> [1.2, 'a', "abc", Some(3)]

```

```
4-element Array{Any,1}:
 1.2
 'a'
 "abc"
 Some{3}
```

但逗号与分号是不能混用的，例如：

```
julia> [1, 2, 3; 4]
ERROR: syntax: unexpected semicolon in array expression
```

心细的读者会发现，创建行向量时，对象的描述是类似于 `1×5 Array{Float64,2}` 这种，但创建列向量时，对象的描述却类似于 `5-element Array{Float64,1}` 这种说明。按照 Julia 官方的说法，行向量实质上仍是二维数组，只不过第一维的阶数为 1 而已，而列向量才是名副其实的一维数组，这从对象描述 `{Float64,2}` 与 `{Float64,1}` 中有区别的维度数也能够看出。

若要创建二维数组（矩阵），将空格与分号综合使用即可，例如：

```
julia> [1 2; 3 4; 5 6]
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6

julia> [1.1 2.2 3.3; 4.4 5.5 6.6]
2×3 Array{Float64,2}:
 1.1  2.2  3.3
 4.4  5.5  6.6
```



注意 在创建矩阵时，不能使用逗号区分不同的行，只能使用分号。故此，笔者建议在罗列方式中尽量用分号，不用逗号，以减少不必要的复杂度。

至此，我们将这种方括号罗列的形式总结一下：首先，方括号用于界定数组的定义范围；其次，不同的行以分号标识；最后，同一行的元素以空格隔开。形式如下：

```
[ A11 A12 ... A1n;           # 省略号表示可以有更多的元素
  A21 A22 ... A2n;
  ...
  Am1 Am2 ... Amn]
```

该形式会创建 $m \times n$ 矩阵。

在采用这种方式创建数组时，如果不让 Julia 自动识别元素类型，可在上述罗列的基础上附加一个类型前缀，以做类型限定。例如：

```
julia> Int32[1 2; 3 4; 5 6]
3×2 Array{Int32,2}:
 1  2
 3  4
 5  6

julia> Float32[1 2 3 4]
1×4 Array{Float32,2}:
 1.0  2.0  3.0  4.0
```

但是,若元素中存在不可精确转换的类型时会出错,例如:

```
julia> Int64[1.2 3 4.5]
ERROR: InexactError: Int64{Int64, 1.2}
```

这点需要开发者注意。

如果元素中存在数组对象(数组嵌套),Julia 会自动将内部的数组展开。例如:

```
julia> [[1 2] 3 [4 5]]
1×5 Array{Int64,2}:
 1  2  3  4  5
```

```
julia> [1 2 3 [4 5]]
1×5 Array{Int64,2}:
 1  2  3  4  5
```

```
julia> [1; 2; [3; 4]]
4-element Array{Int64,1}:
 1
 2
 3
 4
```

但这种情况出现时,内外数组在维度上必须能够兼容,否则会报错,例如:

```
julia> [1 2 [3;4]]
ERROR: DimensionMismatch("mismatch in dimension 1 (expected 1 got 2)")
```

```
julia> [1;2 [3;4]]
ERROR: DimensionMismatch("mismatch in dimension 1 (expected 1 got 2)")
```

事实上,通过罗列方式创建数组,等效于内部调用了 `vcat()` 函数、`hcat()` 函数或 `hvcat()` 函数,它们的对应关系如表 8-2 所示。

表 8-2 数组罗列式创建函数

表 达 式	对 应 函 数	描 述
<code>[A; B; C; ...]</code>	<code>vcat(A,B,C,...)</code>	创建 N 阶列向量
<code>[A B C ...]</code>	<code>hcat(A,B,C,...)</code>	创建 N 阶行向量
<code>[A B; C D; ...]</code>	<code>hvcat(2, A,B,C,D,...)</code>	创建 M×2 阶矩阵,参数 2 指定列数

这三个函数主要用于一维与二维数组的情况,可将其参数串联成数组的元素,构造出对应的数组对象。举例如下:

```
julia> hcat(1, 2, 3, 4)
1×4 Array{Int64,2}:
 1  2  3  4
```

```
julia> vcat(1, 2, 3, 4)
4-element Array{Int64,1}:
 1
 2
 3
 4
```

函数 `hvcat()` 与其他两个函数略有不同的地方是，其第一个参数用于指定矩阵的列数，后面的参数才是元素列表。例如：

```
julia> hvcat(2, 1, 2, 3, 4, 5, 6) # 第一个参数2指定了列数
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

如果改变第一个参数，则会获得不同的矩阵，例如：

```
julia> hvcat(3, 1,2,3,4,5,6)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

但该函数的列数参数必须与后续提供的元素列表大小兼容，否则会报错，例如：

```
julia> hvcat(7, 1,2,3,4,5,6)
ERROR: ArgumentError: number of arrays 6 is not a multiple of the requested
number of block columns 7

julia> hvcat(4, 1,2,3,4,5,6)
ERROR: ArgumentError: number of arrays 6 is not a multiple of the requested
number of block columns 4
```

上例中虽然错误类型一致但原因不同：前者是因为列数超出了元素个数，根本无法实现；后者则是因为 6 个元素无法被 4 整除，即无法分配到 4 列中去。这点在使用中需要注意。

8.1.2 辅助构造函数

除了 `vcat()` 等三个函数之外，Julia 还提供了其他有用的函数用来辅助创建一些特别的数组，有些还能够在创建时直接进行初始化，如表 8-3 所示。

表 8-3 数组辅助构造函数

函 数	描 述
<code>zeros(T, dims...)</code>	元素值全初始化为 0 的数组，元素类型为 T
<code>ones(T, dims...)</code>	元素值全初始化为 1 的数组，元素类型为 T
<code>true(dims...)</code>	元素值全初始化为 true 的 BitArray 类型的数组
<code>false(dims...)</code>	元素值全为 false 的 BitArray 类型的数组
<code>rand(T, dims...)</code>	随机数数组，独立同分布 (iid) 且是均匀分布
<code>randn(T, dims...)</code>	随机数数组，独立同分布 (iid) 且是正态分布
<code>Array{UniformScaling{T}}(v, dims...)</code>	元素为 T 取值为 v 的对角矩阵 (v=1 时为单位矩阵)

表中参数 T 用于指定元素类型，dims 则用于指定各维度的阶数。下面给出一些使用的例子：


```

julia> A1 = zeros{Int64, 2, 3} # 创建2行3列元素值均被初始化为0（类型Int64）的矩阵
2×3 Array{Int64,2}:
 0  0  0
 0  0  0

julia> A2 = zeros{Float64, (2,3)} # 创建2行3列元素值均被初始化为0（类型Int64）的矩阵，效果同A1
2×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0

julia> B1 = ones{Rational, 2, 3} # 创建2行3列元素值均被初始化为1（类型Rational）的矩阵
2×3 Array{Rational{Int64},2}:
 1//1  1//1  1//1
 1//1  1//1  1//1

julia> trues((2,3)) # 创建2行3列、元素均为true的矩阵
2×3 BitArray{2}:
 true  true  true
 true  true  true

julia> falses(size(B1)) # 创建维度及阶数与B2一致的、元素均为false的多维数组
2×3 BitArray{2}:
 false false false
 false false false

julia> C1 = rand{Float64, 2, 3} # 创建2行3列元素类型为Float64的矩阵（均匀分布）
2×3 Array{Float64,2}:
 0.491897  0.162443  0.286046
 0.991597  0.749361  0.200681

julia> C2 = randn{Float64, (2,3)} # 创建2行3列元素类型为Float64的矩阵（正态分布）
2×3 Array{Float64,2}:
 0.10477  -0.367155  -0.00438399
 0.878091  0.977091  1.15638

julia> Array{UniformScaling{Float32}}(1), 2, 3 # 创建对角值取1.0元素类型为Float32的二维数组
2×3 Array{Float32,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0

julia> Matrix{UniformScaling{Int}}(3), (2,3) # 创建对角值取3元素类型为Int的矩阵（二维数组）
2×3 Array{Int64,2}:
 3  0  0
 0  3  0

```

这些辅助函数能够快速地构造出特定的矩阵，在科学计算中会非常方便。

8.1.3 范围表达式

对于一些元素间具有特定后继关系的数组（例如等差数列），Julia 还提供了另外一种方式构造，并提供了一种特定的类型进行容纳，即范围表达式。该类型除了能够完全作为数组使用外，还有广泛的用途。为了避免文字歧义，我们不再使用范围一词，而使用 Range 作为该类型的称谓。

1. 定义方式

可以通过 `range()` 函数创建 `Range` 对象，其调用原型为：

```
range(start; length, stop, step=1)
```

参数中的 `step` 等价于数学中数列相邻两值间的差值（后一个减去前一个），即“公差”，取值可正可负但不能为 0（请留意此点），若不指定默认为 1；参数 `start` 指定该数列计算的起始值，将会出现在数列的首位；而 `stop` 是该数列按照公差进行递进计算所能取到的最远边界值，但因公差限制，故其未必会在该数列中出现；`length` 参数则指定了序列的长度。参数 `length` 与 `stop` 均用于对 `Range` 对象中元素个数的控制，但策略有所不同：后者对个数的要求更为明确，但无法限定序列的取值范围，可根据需求提供这两个键值参数。

该函数在使用示例如下所示：

```
julia> range(1, stop=6)          # 起始值为1, 边界值为6, step默认为1
1:6                             # Range对象的表达方式, start:step:stop, step=1时会省略

julia> foreach(println, ans)    # 逐元素打印
1
2
3
4
5
6

julia> range(1; stop=6, step=2) # 起始值为1, 边界值为6, step取2
1:2:5                           # Range对象的表达方式, start:step:stop, step=2时所以未省略

julia> foreach(println, ans)    # 实际结果中并未取到6
1
3
5
```

其中第二个 `Range` 对象并没有取到 6，这是因为公差为 2 而起始为 1，所以在计算中总会是奇数，而能取得的在 `stop=6` 控制范围的有效值正是 5。再看下面的例子：

```
julia> range(1; length=6)
1:6                             # 因为step=1, 所以得到的stop也是6

julia> range(1; length=6, step=2)
1:2:11                          # 起始为1, step为2, 但stop根据前两个值进行了自动计算,
                                # 取11以保证该Range对象有6个元素
```

事实上，上例中出现的冒号表达也是构造 `Range` 对象的另外一种方式，而且更为简洁，基本表述方法可描述为：

起始值 : 边界值

或者

起始值 : 公差 : 边界值

无论采用哪种方式，创建的 `Range` 对象不会自动展开为逐元素存储的数据序列，而只是记录了等差数列的结构信息，但仍可作为迭代器（可迭代数集）使用。例如：

```
julia> for i = 1:2:6      # 其中边界值5不会取到
    println(i)
end
1 3 5
```

而且, Range 对象也支持浮点数或负数等各种取值, 例如:

```
julia> for i ∈ 0.2:1:5.2
    print(i, " ")
end
0.2 1.2 2.2 3.2 4.2 5.2

julia> for i in 1:-0.75:-1
    print(i, " ")
end
1.0 0.25 -0.5
```

如果需要 Range 对象中数据序列的所有元素具有独立的存取空间, 可使用 `collect()` 将其转换为向量 (一维数组)。例如:

```
julia> collect(1:2:6)
3-element Array{Int64,1}:
 1
 3
 5
```

若是 Range 对象记录的结构信息并不能推导出任何元素, `collect()` 会生成空数组, 例如:

```
julia> 1:2:0
1:2:0

julia> collect(1:2:0)
0-element Array{Int64,1}
```

Range 类型可以表达任意长度的等差数列, 但又不会随着长度的增加占用越来越多的内存, 而且其元素的迭代非常高效, 所以经常用来对数组进行索引或者直接作为迭代数集使用。而且 Range 这种迭代性结构很容易扩展, 一般来说, 对于支持加减操作的类型, 都可以提供 Range 式的操作。

2. 内部结构

事实上, 无公差或有公差时创建的对象类型是有所区别的, 例如:

```
julia> dump(1:3)
UnitRange{Int64}
  start: Int64 1
  stop: Int64 3

julia> dump(1:2:8)
StepRange{Int64,Int64}
  start: Int64 1
  step: Int64 2
  stop: Int64 7      # 虽然指定边界值为8, 但内部会被修正为可取得的尾值
```

其中, 无公差与有公差对象分别为 `UnitRange{Int64}` 与 `StepRange{Int64,Int64}` 两种类型。两者均是参数化复合类型的具象实例, 各自的原型相当于为:

```

struct UnitRange{T<:Real} <: AbstractUnitRange{T}
    start::T
    stop::T
end

```

以及

```

struct StepRange{T, S} <: OrdinalRange{T, S}
    start::T
    step::S
    stop::T
end

```

能够发现, 两者都是不可变结构, 一旦创建便不能修改。

成员中, `start` 和 `stop` 的类型要求一致, `step` 却可以是不同的类型; 其中的 `StepRange` 没有明确的类型限界, 不过 `UnitRange` 限定了类参上界为 `Real` 型, 其继承路径为:

```

AbstractUnitRange{T<:Real} <: OrdinalRange{T,T} <: AbstractRange{T} <:
    AbstractArray{T,1}

```

可见, `UnitRange` 与 `StepRange` 有共同的父类型, 均是 `OrdinalRange`, 区别在于 `UnitRange` 因为没有 `step` 成员。

在结构元素为浮点数时, 还会出现另外一个 `Range` 类型。例如:

```

julia> dump(0.2:1:5.2)
StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}
  ref: Base.TwicePrecision{Float64}
    hi: Float64 0.2
    lo: Float64 -1.1102230246251566e-17
  step: Base.TwicePrecision{Float64}
    hi: Float64 1.0
    lo: Float64 0.0
  len: Int64 6
  offset: Int64 1

```

其中, `Base.TwicePrecision` 因篇幅不展开解释, 仅查看 `StepRangeLen` 类型的基本结构, 即形如:

```

struct StepRangeLen{T, R, S} <: AbstractRange{T}
    ref::R
    step::S
    len::Int64
    offset::Int64
end

```

可见, `StepRangeLen` 同样也是不可变的复合类型, 而且与 `UnitRange`、`StepRange` 有着共同的父类型 `AbstractRange` 及 `AbstractArray{T,1}`。所以 `AbstractArray` 的一些操作也同样适用, 例如, `length()` 取得元素个数, `eltype()` 取得内部的元素类型等等。

对于一个数列生成来说, 这么多的类型似乎显得有些复杂, 实际上这些细节不会影响 `Range` 被广泛使用。其不但具有数组的特性, 而且占据空间不会随着范围的变化而扩张, 还是高效的迭代器。

8.1.4 推导式

除了上述的方法外，推导式 (Comprehensions) 是 Julia 提供的另外一种较为常用的创建数组的方式，形如：

```
[f(x,y,...) for x=rx, y=ry, ...]
```

其中，for 之后的实际是遍历表达式，而且能够在元素生成时通过表达式 $f(x,y,\dots)$ 进行转换计算。这种方式下，数组的维度由遍历表达式中迭代结构的数量决定，阶数由每个迭代数集的大小决定。

先给出一个创建向量的例子：

```
julia> V = rand(4)
4-element Array{Float64,1}:
 0.543331
 0.692199
 0.510864
 0.90449

julia> [ 0.25*V[i-1] + 0.5*V[i] + 0.25*V[i+1] for i=2:length(V)-1 ]
2-element Array{Float64,1}:
 0.609648
 0.654605
```

其中的数组元素由序列 V 相邻三个元素的加权平均计算得到。

当然也可以提供多个迭代结构，以生成高维的数组，例如：

```
julia> [i*j for i=1:3, j=3:5] # 两个迭代结构，所以生成二维数组
3×3 Array{Int64,2}:
 3  4  5
 6  8 10
 9 12 15

julia> [i^2+2j-1.3z for i=1:2, j∈2:5, z in 1:3] # 三个迭代结构，所以生成三维数组
2×4×3 Array{Float64,3}:
[:, :, 1] =
 3.7  5.7  7.7  9.7
 6.7  8.7 10.7 12.7

[:, :, 2] =
 2.4  4.4  6.4  8.4
 5.4  7.4  9.4 11.4

[:, :, 3] =
 1.1  3.1  5.1  7.1
 4.1  6.1  8.1 10.1
```

另外，推导式中的多迭代结构可以使用多个 for 循环单独表示，而且迭代变量能够在后面的迭代结构中使用。例如：

```
julia> [(i,j) for i=1:3 for j=1:i]
6-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (2, 1)
 (2, 2)
 (3, 1)
 (3, 2)
 (3, 3)
```

```
(2, 2)
(3, 1)
(3, 2)
(3, 3)
```

此时的多迭代结构不再生成多维数组，而总会生成一维数组。

此外，在推导式中还可以使用 if 结构对生成的值进行过滤，例如：

```
julia> [(i, j) for i=1:3 for j=1:i if i+j == 4]
2-element Array{Tuple{Int64,Int64},1}:
 (2, 2)
 (3, 1)
```

不过，上述推导式生成数组的元素类型是内部自动确定的。如果要指定类型，则可显式地在推导式之前给出限定类型，例如：

```
julia> Float32[i*j for i=1:3, j=3:5]
3×3 Array{Float32,2}:
 3.0  4.0  5.0
 6.0  8.0 10.0
 9.0 12.0 15.0
```

在使用推导式的过程中，如果没有提供方括号，则会创建一个生成器（Generator）对象。生成器不会预先分配内存并存储元素的值，而是可迭代的结构，只在需要的时候产生数值。例如，下面的表达式能够不占用内存而完成所需序列值的累加：

```
julia> @time sum(1/n^2 for n=1:100)
0.025788 seconds (14.86 k allocations: 792.551 KiB)
1.6349839001848923

julia> @time sum(1/n^2 for n=1:100000)
0.024684 seconds (14.86 k allocations: 792.020 KiB)
1.6449240668982423
```

其中，元素数量即使增加了 1000 倍，内存大小并没有明显的变化。

可见，Julia 提供的推导式功能极为强大，而且使用也极为灵活，能够在开发中带来很大的便利。

8.2 索引访问

在 Julia 中，对数组的访问同样有多种方法，本节将详细介绍。但在此之前，我们需要了解 Julia 数组中两个非常重要的特点：

- ❑ 索引起始下标是 1 而不是 0。这是 Julia 与其他许多语言不同的地方：指向 Julia 数组中第一个元素的索引值不是 0 而是 1（1-based）。这一点务必切记。
- ❑ 数组的数据是列式存储的。一般而言，数组中的数据会存储在连续的内存中。不过在连续存储时存在按行还是按列两种选择。在这个问题上，Julia 采纳的是后者，即列式存储方式。

1. 笛卡尔索引

笛卡尔索引 (Cartesian Indexing) 是最为常见的访问数组的方式：在数组名称之后的方括号中以逗号列出各维度上的索引下标值，即对某个 N 维数组 A ，以某个元素在各维上的具体位置值组合的形式 $A[d_1, d_2, \dots, d_n]$ 获得其取值。例如：

```
julia> a = [1 2 3 4 5 6 7]
1×7 Array{Int64,2}:
 1  2  3  4  5  6  7

julia> a[1]           # 第1个元素
1

julia> a[4]           # 第4个元素
4

julia> b = rand(2, 3)
2×3 Array{Float64,2}:
 0.133385  0.0484837  0.187336
 0.80238  0.534638  0.758194

julia> b[1, 2]        # 第1行第2列元素
0.04848366285577255

julia> b[2, 3]        # 第2行第3列元素
0.7581943142021761
```

这种索引方式对前文介绍的 Range 对象同样适用，例如：

```
julia> j = 1:0.3:3;

julia> j[1]
1.0

julia> j[4]
1.9

julia> j[7]
2.8
```

另外，如果给定的索引下标超出了维度的阶数，则会导致越界错误，例如：

```
julia> a[8]
ERROR: BoundsError: attempt to access 1×7 Array{Int64,2} at index [8]

julia> b[3, 2]
ERROR: BoundsError: attempt to access 2×3 Array{Float64,2} at index [3, 2]
```

在开发中，可以通过数组属性函数 `size()` 先取得各维度阶数，再以合理的下标值对数组进行访问。

2. end 操作

为了操作的便利，尤其是对如尾元素这种边界值的访问，Julia 提供了 `end` 关键字，能够自动取得对应维度上的尾元素索引值。例如：

```
julia> c = [1 0 2; 3 0 5]
2×3 Array{Int64,2}:
 1  0  2
 3  0  5

julia> a[end]          # 相当于a[7]
7

julia> c[end, end]     # 相当于c[2,3]
5
```

而且在索引表达式中还能够以 `end` 为基础，通过适当的计算实现对数集的灵活访问：

```
julia> a[(2*end)>>2]
3

julia> c[end-1, end-2]
1
```

3. 赋值修改

在获得元素后，便可对其进行修改操作，例如：

```
julia> a[3] = 30
30

julia> b[2, 3] = 25
25
```

赋值后会返回被赋予的值，而且原始数据的内部会被改变：

```
julia> a
1×7 Array{Int64,2}:
 1  2 30  4  5  6  7

julia> b
2×3 Array{Float64,2}:
 0.133385  0.0484837  0.187336
 0.80238  0.534638   25.0
```

但如果将一个无法转换到数组元素类型的值赋予某个元素，则会报错：

```
julia> a[4] = 3.8
ERROR: InexactError: Int64{Int64, 3.8}
```

这是因为数组 `a` 的元素类型是 `Int64`，欲赋值的 `3.8` 则为 `Float64` 类型，无法直接转换。除此之外，对于只读的结构也是无法进行修改赋值的，例如：

```
julia> j[2] = 3
ERROR: setindex! not defined for StepRangeLen{Float64,Base.TwicePrecision{Float64},
Base.TwicePrecision{Float64}}
```

这是因为元组是不可变的。

4. CartesianIndex

对于维度很高的数组，比如 10 维、20 维甚至更高维，要以数值罗列的方式给出索引的下标值，还是不方便。我们需要对索引进行更多灵活的操作。为此，Julia 提供了

`CartesianIndex` 类型，专门用于封装索引下标，其构造方法原型为：

```
CartesianIndex{N}(i, j, k...)
CartesianIndex{N}((i, j, k...))
```

其中，`i`，`j`，`k` 指定需要访问的高维数组上各个维度上的索引下标，参数 `N` 限定坐标的数量（维度数）。坐标参数可以接收下标值组合而成的元组，也可以接收每个具体下标作为独立参数。因为坐标参数的个数或元组的长度即可隐含地表达维度值，所以一般可以不提供 `N`，Julia 会自动生成。例如：

```
julia> i = CartesianIndex(1,3)
CartesianIndex{2}((1, 3))

julia> i = CartesianIndex((1,3))
CartesianIndex{2}((1, 3))
```

生成了相同的对象。此后，便可基于此访问对应位置的元素，例如：

```
julia> a[i]
30

julia> b[i]
0.187336

julia> c[i]
2
```

下面看一个维度更高的例子：

```
julia> d = rand(3,4,2,2)
3×4×2×2 Array{Float64,4}:
# 为篇幅，已省略

[:, :, 1, 2] =
 0.468359  0.517037  0.0585288  0.718665
 0.404673  0.818096  0.411517   0.394487
 0.296784  0.154974  0.128332   0.0995659

[:, :, 2, 2] =
 0.0707639  0.605185  0.224987  0.409333
 0.684682   0.624774  0.568148  0.85278
 0.864965   0.297632  0.339797  0.315841

julia> d[2, 3, 1, 2]
0.41151670781400673

julia> i = CartesianIndex(2, 3, 1, 2)
CartesianIndex{4}((2, 3, 1, 2))

julia> d[i]                                # 与d[2, 3, 1, 2]访问的结果一致
0.41151670781400673
```

有了 `CartesianIndex` 类型，我们就可以将繁杂的索引操作进行封装，实现对高维数组的便捷访问或存取。

5. 线性索引

除了上文介绍的笛卡尔索引方式，借助于数组内部的列式内存存储结构，还可以使用线性索引的方式。例如，对数组 A，其内容为：

```
julia> A
3×4 Array{Int64,2}:
 1  4  7 10
 2  5  8 11
 3  6  9 12

julia> A[8]
8
```

使用表达式 A[8] 会取得相当于笛卡尔索引 [2, 3] 位置处的元素值 8。之所以如此，因为从 A 的第一列开始，逐列后移，当移到第 2 行第 3 列的位置时，总计移动了 8 个位置。

对于 Array 这种稠密数组，其数据在内存中占据了一片连续的空间，所以当采用线性索引方式对 Array 的对象进行访问时，实际是将内部的访问指针以数组起始地址作为起点，向后移动了索引值大小的内存偏移。

在访问数组时，如果要访问尾部元素，除了使用类似 A[length(A)] 这种方式外，还可以使用 A[lastindex(A)]。事实上，前文的 end 关键字在底层便相当于 lastindex() 函数。该函数不但能够获得线性索引的最大值，也可以通过其可选的第二个参数获得指定维度上的最大索引值，例如：

```
julia> lastindex(A)      # 最大线性索引值
12

julia> lastindex(A, 1)   # 行索引最大值
3

julia> lastindex(A, 2)   # 列索引最大值
4
```

6. 索引转换

在 Julia 中主要有两种索引方式，便是上文介绍的笛卡尔索引与线性索引。新版的 Julia 中不再支持两种索引方式的混合，但是支持通过 LinearIndices() 和 CartesianIndices() 两个函数对它们进行相互的转换。前者会生成输入数组同结构的数组，而每个元素中的值便是输入数组中各元素的线性索引值；类似地，后者同样会得到输入数组同结构的新数组，而元素便是输入数组各元素的笛卡尔索引，是一个 CartesianIndex 对象，而待转换的线性索引作为其下标访问便可得到对应的笛卡尔坐标。为了更好地理解，看下面一个例子：

```
julia> B = [10 13 16 19; 11 14 17 20; 12 15 18 21]
3×4 Array{Int64,2}:
10 13 16 19
11 14 17 20
12 15 18 21
```

先使用 LinearIndices() 获得该数组每个元素对应的线性索引值：

```
julia> bli = LinearIndices(B)
3×4 LinearIndices{2,Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}}:
 1  4  7 10
 2  5  8 11
 3  6  9 12
# B中第一行元素的线性索引值分别对应1、4、7和10
```

此后，便可以基于 bli 获得任意某个笛卡尔坐标的线性索引，例如：

```
julia> bli[2, 3]
8
# 笛卡尔坐标 (2, 3) 对应线性坐标8

julia> B[2,3] == B[8]
true
# 分别用两种索引访问，元素一致

julia> B[3,2] == B[bli[3,2]]
true
# 两种方式方位3行2列对象的元素，测试是否一致
```

下面我们再看如果进行反向的转换。首先，使用 CartesianIndices() 提取 B 中各元素的笛卡尔坐标集合，如下所示：

```
julia> bci = CartesianIndices(B)
3×4 CartesianIndices{2,Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}}:
 CartesianIndex(1, 1) CartesianIndex(1, 2) CartesianIndex(1, 3) CartesianIndex(1, 4)
 CartesianIndex(2, 1) CartesianIndex(2, 2) CartesianIndex(2, 3) CartesianIndex(2, 4)
 CartesianIndex(3, 1) CartesianIndex(3, 2) CartesianIndex(3, 3) CartesianIndex(3, 4)
```

然后，将需要转换的线性索引值作为该 CartesianIndices 对象的下标，即可得到其笛卡尔坐标了，即：

```
julia> bci[8]
CartesianIndex{2, 3}
# 线性索引8对应笛卡尔坐标 (2, 3)

julia> B[CartesianIndex(2, 3)] == B[8]
true
# 两者对应同一个元素

julia> B[6] == B[bci[6]] == 15
true
# 两种方式获得的元素均为15，即 (3, 2) 对应的值
```

当然，这种转换并不是一定依赖于已有的数组，也可以直接输入维度信息，以对其进行转换。例如：

```
julia> cartesian = CartesianIndices((1:3, 1:2))
3×2 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex(1, 1) CartesianIndex(1, 2)
 CartesianIndex(2, 1) CartesianIndex(2, 2)
 CartesianIndex(3, 1) CartesianIndex(3, 2)
# 生成行范围1:3，列范围1:2的笛卡尔坐标集合

julia> cartesian[4]
CartesianIndex{1, 2}
# 取得该笛卡尔坐标集，线性索引为4的坐标
```

再例如：

```
julia> linear = LinearIndices((1:3, 1:2))
3×2 LinearIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 1  4
 2  5
 3  6
# 生成行范围1:3，列范围1:2的线性索引集合
```

```
julia> linear[1,2]                                # 取得笛卡尔坐标(1,2)的线性索引值
4
```

上述的转换方式巧妙地利用了 Julia 内置的数组结构，能够方便地建立线性索引与笛卡尔坐标之间的对应关系，而且能够进行边界的检查，避免越界错误。

7. 结构转换

除了索引转换方式外，Julia 提供了另外一种机制。我们可以通过 `vec()` 函数或 `[:]` 表达式将多维数组转为线性结构（列向量），而且只是引用操作，不会复制数据创建新数组。所以对向量修改时，源数组也会同时变更。例如：

```
julia> p = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> q = vec(p)
6-element Array{Int64,1}:
 1
 4
 2
 5
 3
 6
```

修改 `q` 中一个元素，会发现：

```
julia> q[3] = 10;

julia> p
2×3 Array{Int64,2}:
 1 10  3
 4  5  6
```

可见其中 `p` 的元素也同时被修改，实际上：

```
julia> CartesianIndices((1:2, 1:3))[3]
CartesianIndex{2}(1, 2)
```

即线性索引 3 正对应源数组笛卡尔索引 (1, 2) 的位置。但需要注意的是，如果使用 `q = p[:]` 将 `p` 转为向量，会生成新的对象，在新对象所作出的修改并不会传递到原始数据中。

除了上述的方法外，可以使用更为强大的 `reshape()` 函数，将输入数组（源数据）的结构调整为需要的维度和阶数。与 `vec()` 类似，新结构中的数据是输入数组的引用，对新结构的修改同样会作用到原数据中。例如：

```
julia> m = reshape(p, 3, 2)
3×2 Array{Int64,2}:
 1  5
 4  3
10  6

julia> m[3,1] = 20;

julia> p
```



```
2×3 Array{Int64,2}:
 1  20  3
 4   5  6
```

但在该函数的使用过程中，需确保新结构的元素个数与源结构一致。为此可以在某些维度确定后，留下一个维度使用：标识符提供，这样 Julia 内部能够自动计算该维度阶数，使得总长度保持一致。例如：

```
julia> n = reshape(p, 3, :)
3×2 Array{Int64,2}:
 1  5
 4  3
20  6
```

```
julia> n[2,2] = 30
30
```

```
julia> p
2×3 Array{Int64,2}:
 1  20  30
 4   5   6
```

8.3 遍历迭代

所谓遍历，只在一次处理中对数集中的所有元素完成一次且仅一次的访问。对于数组而言，主要有两种方案：一种是获得索引集再通过其逐一访问数集中的元素；另一种是直接将数组作为可迭代数集，逐一获得其中的元素。

1. 索引循环

一般而言，遍历因为需要访问所有的元素，所以需要使用循环结构，例如 while 与 for 结构。

先看一个 while 循环的例子。先通过 size() 获得数组的维度情况，然后通过循环控制变量逐一获得索引值，再逐一访问对应的元素值。例如：

```
julia> a = reshape(collect(1:12), 3, 4)
3×4 Array{Int64,2}:
 1  4  7 10
 2  5  8 11
 3  6  9 12

julia> size(a)
(3, 4)

julia> i = 1;
julia> while i <= 3
    j = 1;
    while j <= 4
        print(a[i,j], " ")
        j += 1
    end
    i += 1
end
```

对行索引循环
重置到新行的第1列
对列索引循环
取得对应位置的元素
列索引递增

```
println("")          # 换行打印
global i += 1        # 移动到下一行索引
end
```

运行的结果为：

```
1 4 7 10
2 5 8 11
3 6 9 12
```

这种自行写循环并控制索引不断递增的方式，往往需要考虑很多因素，比如变量的作用域、维度上界变化、维度越界的控制等，实际的工程代码会比上例中的考虑更多。

再看一个使用 for 循环的例子，例如：

```
julia> for i = 1:size(a,1)
    for j = 1:size(a,2)
        print(a[i,j], " ")
    end
    println("")
end
1 4 7 10
2 5 8 11
3 6 9 12
```

也可以利用 for 循环的特点，同时对多维进行遍历，减少循环结构，如下：

```
julia> for i=1:3, j=1:4
    print(a[i, j], " ")
end
1 4 7 10 2 5 8 11 3 6 9 12
```

至于如何在复合遍历的情况下实现前例中的分行效果，有兴趣的读者可以做些尝试。

2. 索引序列

采用多维索引对数组遍历时，除了像上述那种自己拼装出笛卡尔下标索引外，也可借助 Julia 提供的 `CartesianIndices` 对象实现遍历功能。该类型能够根据指定的坐标范围构造出笛卡尔坐标迭代器。例如：

```
julia> r = CartesianIndices{1:3, 1:4}
3×4 CartesianIndices{2,Tuple{UnitRange{Int64},UnitRange{Int64}}}:
 CartesianIndex{1, 1} CartesianIndex{1, 2} CartesianIndex{1, 3} CartesianIndex{1, 4}
 CartesianIndex{2, 1} CartesianIndex{2, 2} CartesianIndex{2, 3} CartesianIndex{2, 4}
 CartesianIndex{3, 1} CartesianIndex{3, 2} CartesianIndex{3, 3} CartesianIndex{3, 4}
```

这样，我们便可以通过遍历 `CartesianIndices` 对象对数组进行访问了，即：

```
julia> for k in r
    println(k[1], ",", k[2], " is ", a[k])
end
1,1 is 1
2,1 is 2
3,1 is 3
1,2 is 4
2,2 is 5
3,2 is 6
1,3 is 7
```

```

2,3 is 8
3,3 is 9
1,4 is 10
2,4 is 11
3,4 is 12

```

可见，循环结构变得更简洁了。

3. 线性遍历

上述通过索引方式进行遍历时，依据的是笛卡尔索引。如果在处理中不关心元素在数组内部的位置，则通过线性索引方式迭代遍历更为方便。例如：

```

julia> for i = 1:lastindex(a)
           print(a[i], " ")
       end
1 2 3 4 5 6 7 8 9 10 11 12

```

或者：

```

julia> for i = 1:length(a)
           print(a[i], " ")
       end
1 2 3 4 5 6 7 8 9 10 11 12

```

其中，`length()` 函数用于返回数组的元素总个数。

此外，也可以通过 `eachindex()` 函数自动生成数组的线性索引迭代器，也可以用于元素的遍历。例如：

```

julia> for i in eachindex(a)
           print(a[i], " ")
       end
1 2 3 4 5 6 7 8 9 10 11 12

```

效果与上例相当。

4. 元素迭代

如 5.3 节所述，数组同样是一种可以迭代的数集，可以通过 `for` 循环利用遍历表达式，按序直接获得其中的元素。例如：

```

julia> for x in a
           print(x, " ")
       end
1 2 3 4 5 6 7 8 9 10 11 12

```

或者：

```

julia> for x = a
           print(x, " ")
       end
1 2 3 4 5 6 7 8 9 10 11 12

julia> for x ∈ a
           print(x, " ")
       end
1 2 3 4 5 6 7 8 9 10 11 12

```

这种方式同样是按列线性访问的，在一些不需要下标索引值参与处理的过程中，会比较方便，也更为简洁。

8.4 子数组与视图

有时候，需要对数组的某个局部进行批量操作，采用循环迭代的方式显然会比较麻烦，因为需要对索引边界进行准确的控制。Julia 提供了比较丰富的子数组操作，而且为了避免过多的内存操作，还提供了视图机制对数组的局部进行高效的存取。

8.4.1 范围切片

取得子数组的一种直接方式是，以范围表达式作为索引提取需要的部分。例如：

```
julia> a = reshape(collect(1:12), 3, 4)
3×4 Array{Int64,2}:
 1  4  7 10
 2  5  8 11
 3  6  9 12
```

```
julia> a[2:3, 3:4]
2×2 Array{Int64,2}:
 8 11
 9 12
```

但是，只提取了单行或单列会构造为向量，例如：

```
julia> a[2, 2:end]
3-element Array{Int64,1}:
 5
 8
11
```

而且，在索引结构中通过对子部分索引进行重排，可以调整元素在新建子数组中的位置：

```
julia> a[2, [2 3; 4 1]]
2×2 Array{Int64,2}:
 5  8
11  2
```

其中，第一维指定要取得源数组 `a` 中的第 2 行；第二维通过数组结构给出了第 2 行中感兴趣的列并对列序进行了重排，而且实现了二维的结构。

如果对某个维度整体感兴趣，而且无须变更结构，则可使用操作符 `:` 进行整维度的提取。例如：

```
julia> a[3, :]
4-element Array{Int64,1}:
 3
 6
 9
12

julia> a[:, 3]
3-element Array{Int64,1}:
 7
 8
 9
```



```
3-element Array{Int64,1}:
 7
 8
 9
```

8.4.2 逻辑索引

对于感兴趣的元素，除了通过索引指定，也可以通过“逻辑索引”（Logical Indexing）的方式实现。在这种方式中，在对应维度上提供长度一致的 Bool 型数组便能够过滤掉不需要的元素。例如：

```
julia> a[[true,false,true], 2]
2-element Array{Int64,1}:
 4
 6
```

其中，数组 a 第 2 列中的元素 5 对应的第 2 行设置为 false，所以被过滤掉了。当然，也可以同时所有维度上进行逻辑索引，例如：

```
julia> a[[true,false,true], [true,true,true,false]]
2×3 Array{Int64,2}:
 1  4  7
 3  6  9
```

在这种方式中，各维度上的逻辑值有任意一个为 false 便会被过滤掉。

采用这种方式，我们便能够通过外部的变量或因素控制元素的提取。例如：

```
julia> x=2;

julia> a[[iseven(x), false, !isodd(x)], 2]
2-element Array{Int64,1}:
 4
 6
```

可见，对 a 的行进行提取时，第 1 行与第 3 行是否被过滤掉依赖于 x 是奇数还是偶数。

还有一个通过条件索引的方式，能够过滤满足条件的元素，而且像上述那样需直接提供与元素个数的布尔序列。例如：

```
julia> a[a .> 8]
4-element Array{Int64,1}:
 9
10
11
12
```

不过这种方式会将原数组视作一维序列，无法针对特定的维度进行过滤，而且输出的也是一维数组。

8.4.3 局部视图

通过上节介绍的索引方式取得子数组时，一般需要将原始数据复制到新的数组对象中。如果数组很大，这种复制操作会带来性能问题。为此，Julia 提供了一种视图机制，能够避



免数据复制操作。

1. 创建视图

创建数组视图的函数为：

```
view(A, inds...)
```

其中 `inds` 为可变参数，用于数组 `A` 中感兴趣部分的索引。

调用该函数时，会返回以源数组为基础的 `SubArray` 类型的对象（关于 `SubArray` 可参考官方资料，本书不作介绍）。例如：

```
julia> a = reshape(collect(1:12), 3, 4)
3×4 Array{Int64,2}:
 1  4  7 10
 2  5  8 11
 3  6  9 12

julia> b = view(a, 2:3, 2:3)
2×2 view{::Array{Int64,2}, 2:3, 2:3} with eltype Int64:
 5  8
 6  9

julia> b[1,1]
5
```

可见，创建的视图对象，能够采用数组索引的方式进行访问，没有什么特别之处。但需注意的是，视图的索引空间是独立的，不依赖于源数组。

如上面所示，视图只是原数据的引用，与其一同指向了同一个数据区，所以对视图的修改更新操作同样会传递到源数组。例如：

```
julia> b[1,1] = 200;           # 修改视图中的元素

julia> b
2×2 view{::Array{Int64,2}, 2:3, 2:3} with eltype Int64:
200  8           # 视图已发生变化
 6  9

julia> a
3×4 Array{Int64,2}:
 1  4  7 10
 2 200 8 11      # 源数组也跟着发生了变化
 3  6  9 12
```

另外，与 `view()` 函数功能一致的还有 `@view` 宏，其也能够创建视图结构。例如：

```
julia> c = @view a[2:3,2:3]
2×2 view{::Array{Int64,2}, 2:3, 2:3} with eltype Int64:
200  8
 6  9

julia> fill!(c, 300)           # 将视图整体进行修改
2×2 view{::Array{Int64,2}, 2:3, 2:3} with eltype Int64:
300 300
300 300
```



```
julia> a                                     # 被映射的部分全部被更新
3×4 Array{Int64,2}:
 1    4    7   10
 2   300  300  11
 3   300  300  12
```

可见效果是相同的，都能够借助局部的视图修改原数据区被映射的内容。

2. 性能对比

为了说明上述创建子数组的方式与视图方式的差异，对比一下两者的运行效果，代码如下：

```
julia> A = rand(10000,10000);

# 范围索引方式
julia> @time A[1000:5000,1000:5000];
0.258203 seconds (8 allocations: 122.132 MiB, 50.12% gc time)

julia> @time A[1000:5000,1000:5000];
0.267561 seconds (8 allocations: 122.132 MiB, 52.32% gc time)

# 视图方式
julia> @time view(A,1000:5000,1000:5000);
0.000073 seconds (36 allocations: 1.172 KiB)

julia> @time view(A,1000:5000,1000:5000);
0.000070 seconds (36 allocations: 1.172 KiB)

julia> @time view(A,1000:5000,1000:5000);
0.000073 seconds (36 allocations: 1.172 KiB)
```

例中范围索引与视图都取了同一片子数组，但性能上有着巨大的差异：在耗时上，相差竟有 3500 倍之多，而且在内存分配及 GC 方面，视图方式有着极为明显的优势。



提示 建议在开发过程中尽量采用视图的方式对数组的子数组进行操作。

在实践中，如果有必要，可使用 `parent()` 函数获得某个视图对象的源数组。例如：

```
julia> p = parent(c)
3×4 Array{Int64,2}:
 1    4    7   10
 2   300  300  11
 3   300  300  12

julia> p === a
true
```

其中，若参数提供的对象本身不是视图，则直接返回该数组对象。

8.5 稀疏数组

元素大多为零，甚至为零的元素数量远远大于非零元素数量时，这种数组称为稀疏数组 (Sparse Arrays)。对于维度很高、阶数很大的稀疏数组，如果将每个元素都存储在内存



中,甚至占据连续的内存区,会造成资源极大的浪费。为此,人们想到了对这种矩阵进行压缩存取的方法。通过采用特殊的数据结构对这种稀疏数组进行存储,往往能够显著地提高存储与计算效率。

针对向量与矩阵,在 Julia 的 SparseArrays 模块中以压缩稀疏列 (Compressed Sparse Column, CSC) 的方式提供了稀疏结构的支持,并定义了专门的类型:

```
struct SparseMatrixCSC{Tv, Ti <: Integer} <: AbstractSparseMatrix{Tv, Ti}
    m::Int                # 行数
    n::Int                # 列数
    colptr::Vector{Ti}    # 列指针
    rowval::Vector{Ti}    # 存储元素值的行索引序列
    nzval::Vector{Tv}     # 存储的非0元素值
end

struct SparseVector{Tv, Ti <: Integer} <: AbstractSparseVector{Tv, Ti}
    n::Int                # 长度
    nzind::Vector{Ti}     # 有效值的索引
    nzval::Vector{Tv}     # 有效非零值的存储区
end
```

其中的 Tv 标识元素的数据类型,而 Ti 则是用于记录列指针及行索引。至于其中的 Vector 类型,则是 Array{T,1} 的别称。

8.5.1 典型稀疏结构

典型的稀疏矩阵是单位方阵与零矩阵,我们可以直接使用 spzeros() 函数或 sparse() 函数构成出稀疏版的对象,即:

```
julia> using SparseArrays

julia> spzeros(3)                # 3元素的稀疏版零向量
3-element SparseVector{Float64,Int64} with 0 stored entries

julia> spzeros(Float32, 2,3)     # 2×3的稀疏版零矩阵,且指定元素类型为Float32
2×3 SparseMatrixCSC{Float32,Int64} with 0 stored entries

julia> using LinearAlgebra

julia> sparse(I, 3, 5)           # 3×5的稀疏版单位矩阵,I使用前需using LinearAlgebra
3×5 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
 [1, 1]  =  1.0
 [2, 2]  =  1.0
 [3, 3]  =  1.0

julia> sparse(UniformScaling{Int}(10), 3, 4) # 元素类型为Int、对角取值为10的单位矩阵
3×4 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
 [1, 1]  =  10
 [2, 2]  =  10
 [3, 3]  =  10
```

当然,也可以直接使用构造方法创建这种特殊对象,例如:




```
julia> SparseMatrixCSC(I, 3, 4)
3×4 SparseMatrixCSC{Bool,Int64} with 3 stored entries:
 [1, 1]  = true
 [2, 2]  = true
 [3, 3]  = true

julia> SparseMatrixCSC(UniformScaling{Float32}(10), 3, 4)
3×4 SparseMatrixCSC{Float32,Int64} with 3 stored entries:
 [1, 1]  = 10.0
 [2, 2]  = 10.0
 [3, 3]  = 10.0
```

```
julia> SparseMatrixCSC(0.0*I, 3, 4) # 3行4列的元素全部为0的稀疏矩阵
3×4 SparseMatrixCSC{Float64,Int64} with 0 stored entries
```

类似于 `rand()` 及 `randn()` 函数，也可以通过 `sprand()` 及 `sprandn()` 函数生成稀疏版的随机数组。前者的调用原型为：

```
sprand([rng], m, [n], p::AbstractFloat, [rfn], [type])
sprandn([rng], m[,n], p::AbstractFloat)
```

由中括号限定的参数均是可选参数，其中的 `rng` 是随机数生成器对象；`type` 用于指定元素类型；`m` 与 `n` 指定了阶数（若只有 `m` 则创建向量）；`p` 用于指定生成非零值的概率；参数 `rfn` 是随机函数的某个方法，包括 `rand()`、`randn()` 及 `randexp()` 等方法，若省略则默认为均匀分布。后者的区别仅在于省略了 `rfn` 参数，生成的非零数均来自正态分布。

下面看几个例子：

```
julia> sprand(Bool, 2, 2, 0.5) # 生成2行2列的稀疏矩阵，元素类型为Bool，非零概率0.5
2×2 SparseMatrixCSC{Bool,Int64} with 2 stored entries:
 [1, 1]  = true
 [2, 1]  = true

julia> sprand(Float64, 3, 0.75) # 生成3元的稀疏向量，元素类型为Float64，非零概率0.75
3-element SparseVector{Float64,Int64} with 1 stored entry:
 [3] = 0.298614

julia> sprandn(2, 2, 0.75) # 生成2行2列的稀疏矩阵，元素类型默认为Float64，非零概率0.75
2×2 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
 [1, 1]  = 0.586617
 [1, 2]  = 0.297336

# 3行4列稀疏矩阵，非零概率0.3，基于MersenneTwister随机生成器，使用指数随机方法randexp
julia> sprand(MersenneTwister(1234), 3, 4, 0.3, randexp)
3×4 SparseMatrixCSC{Float64,Int64} with 4 stored entries:
 [1, 2]  = 0.678772
 [2, 2]  = 0.0107092
 [3, 2]  = 0.0935253
 [2, 3]  = 1.36906
```

由于随机数的生成，所以上述命令重复运行未必能得到相同的结果。

8.5.2 结构转换

此外，还可以将现有的稠密矩阵转为稀疏结构，例如：



```
julia> A = Matrix{UniformScaling{Float32}}(5),(3,4) # 3行4列的稠密矩阵
3×4 Array{Float32,2}:
 5.0  0.0  0.0  0.0
 0.0  5.0  0.0  0.0
 0.0  0.0  5.0  0.0
```

```
julia> B = [1.0, 0.0, 1.0]
3-element Array{Float64,1}:
 1.0
 0.0
 1.0
```

```
julia> a = sparse(A) # 转为稀疏版
3×5 SparseMatrixCSC{Float64,Int64} with 3 stored entries:
 [1, 1] = 5.0
 [2, 2] = 5.0
 [3, 3] = 5.0
```

```
julia> b = sparse(B)
3-element SparseVector{Float64,Int64} with 2 stored entries:
 [1] = 1.0
 [3] = 1.0
```

其中，`sparse()` 函数会创建新的对象，并将参数中的原始数据复制到稀疏结构中。或者使用稀疏结构的构造方法转换：

```
julia> a = SparseMatrixCSC(A)
3×4 SparseMatrixCSC{Float32,Int64} with 3 stored entries:
 [1, 1] = 5.0
 [2, 2] = 5.0
 [3, 3] = 5.0
```

```
julia> b = SparseVector(B)
3-element SparseVector{Float64,Int64} with 2 stored entries:
 [1] = 1.0
 [3] = 1.0
```

反之，也可以通过函数 `collect()` 将稀疏结构转回常规的稠密结构。例如：

```
julia> AA = collect(a)
3×4 Array{Float32,2}:
 5.0  0.0  0.0  0.0
 0.0  5.0  0.0  0.0
 0.0  0.0  5.0  0.0
```

```
julia> BB = collect(b)
3-element Array{Float64,1}:
 1.0
 0.0
 1.0
```

或者使用稠密数组的类型进行构造，转换到常规的数组对象，即：

```
julia> AA = Matrix(a)
3×4 Array{Float32,2}:
 5.0  0.0  0.0  0.0
 0.0  5.0  0.0  0.0
```



```

0.0  0.0  5.0  0.0

julia> BB = Vector(b)
3-element Array{Float64,1}:
 1.0
 0.0
 1.0

```

如果不确定稀疏对象是否是向量，可以使用 `Array` 进行转换：

```

julia> AA = Array(a)
3×4 Array{Float32,2}:
 5.0  0.0  0.0  0.0
 0.0  5.0  0.0  0.0
 0.0  0.0  5.0  0.0

julia> BB = Array(b)
3-element Array{Float64,1}:
 1.0
 0.0
 1.0

```

若要判断一个数组是否是稀疏的，可用 `issparse()` 函数，例如：

```

julia> issparse(b)
true

julia> issparse(B)
false

```

8.5.3 内容映射

如果基于已有内容构造一个稀疏矩阵，可使用 `sparse()` 函数的另外一个原型，如下所示：

```

sparse(I, J, V, [m, n, combine])    # 构造稀疏矩阵

```

其中，`V` 是欲填入稀疏结构的值向量；向量 `I` 是 `V` 中元素的行索引；向量 `J` 是对应的列索引。使用该函数构造一个 `S` 对象时，应满足 `S[I[k], J[k]] = V[k]`，其中 `k` 为三个向量的位置索引；`I` 中所有的元素均满足 `1 ≤ I[k] ≤ m`，而 `J` 中元素应满足 `1 ≤ J[k] ≤ n`；若 `m` 和 `n` 未指定，会分别默认地设置为 `maximum(I)` 及 `maximum(J)`，即各自的极大值。另一个参数 `combine` 用于指定合并重复索引值的方法，若未指定，当 `V` 元素为布尔型时，默认为 `|` 运算符，否则默认为 `+` 运算符。类似地，下述的构造方法用于构造稀疏向量：

```

sparsevec(I, V, [m, combine])    # 构造稀疏向量

```

对于新对象 `R`，应满足 `R[I[k]] = V[k]` 条件。

上述两个函数在调用时，参数 `I` 或 `J` 实际是描述了目标稀疏结构中非零元素的位置，而参数 `V` 便是这些非零元素，所以笔者把这样的创建过程称为“内容映射”方法。下面举例说明这两个函数的用法，例如：

```

julia> using SparseArrays

```



```
julia> II = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];

# II最大值为5, J最大值为18, 故构造的稀疏矩阵为5行18列
julia> S = sparse(II, J, V)
5×18 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
 [1, 4] = 1
 [4, 7] = 2
 [5, 9] = 3
 [3, 18] = -5

julia> R = sparsevec(II, V)
5-element SparseVector{Int64,Int64} with 4 stored entries:
 [1] = 1
 [3] = -5
 [4] = 2
 [5] = 3

julia> sparsevec([1, 3, 1, 2, 2], [true, true, false, false, false]) # 索引1与2重复
3-element SparseVector{Bool,Int64} with 3 stored entries:
 [1] = true           # 同样索引1对应的true与false取或运算, 得到true
 [2] = false          # 同样索引2对应的false与false取或运算, 得到false
 [3] = true

julia> II = [1, 3, 3, 5]; V = [0.1, 0.2, 0.3, 0.2]; # 索引3重复
julia> sparsevec(II, V)
5-element SparseVector{Float64,Int64} with 3 stored entries:
 [1] = 0.1
 [3] = 0.5             # 同样索引3对应的0.2及0.3被累加
 [5] = 0.2

julia> sparsevec(II, V, 8, -) # 指定了阶数及索引合并的方法
8-element SparseVector{Float64,Int64} with 3 stored entries:
 [1] = 0.1
 [3] = -0.1            # 同样索引3对应的0.2及0.3进行减法计算
 [5] = 0.2
```

再例如:

```
julia> S2 = sparse(II, J, V, 20, 30) # 执行稀疏矩阵为20行30列
20×30 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
 [1, 4] = 1
 [4, 7] = 2
 [5, 9] = 3
 [3, 18] = -5
```

对于稀疏结构, 我们仍可以使用 `size()` 函数查看其维度和阶数信息:

```
julia> size(S)
(5, 18)
```

```
julia> size(S2)
(20, 30)
```

可见, `S2` 的数据虽然仍与 `S` 一致, 但阶数发生了变化。不过, 如果我们使用 `sizeof()` 函数查询两者占据的内存字节数:




```
julia> sizeof(S)
40
```

```
julia> sizeof(S2)
40
```

会发现两者所占空间的并没有发生变化。

我们再看另外一个例子：

```
julia> Is = [1; 2; 3];
```

```
julia> Js = [10; 20; 30];
```

```
julia> Vs = [100, 0, 300];
```

```
julia> M = sparse(Is, Js, Vs)
3×30 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
 [1, 10] = 100
 [2, 20] = 0
 [3, 30] = 300
```

我们可以注意到 `Vs` 中的 0 并没有被自动地去掉，仍保留在稀疏矩阵中。这种由外部数据引入的零值称为结构性零值 (Structural Nonzeros)，如果要丢弃这些零值，可以对稀疏对象调用 `dropzeros!()` 函数：

```
julia> MM = dropzeros!(M)
3×30 SparseMatrixCSC{Int64,Int64} with 2 stored entries:
 [1, 10] = 100
 [3, 30] = 300
```

可以使用 `nnz()` 函数获得稀疏结构中非零元素的个数，例如：

```
julia> nnz(MM)
2
```

```
julia> nnz(M)
2
```

该函数在统计时会忽略其中的结构性零值。如果要得到哪些位置是非零值，可以使用如下的方法：

```
julia> findall(!iszero, M)
2-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 10)
 CartesianIndex(3, 30)
```

```
julia> findall(!iszero, MM)
2-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 10)
 CartesianIndex(3, 30)
```

也可以调用 `findnz()` 函数同时获得非零值的位置及具体的取值，例如：

```
julia> findnz(M)
([1, 3], [10, 30], [100, 300])
```

```
julia> findnz(MM)
([1, 3], [10, 30], [100, 300])
```



结果是元组结构，最后一个元素为非零值，前面的是这些非零值在各维度上的索引值。

在对稀疏数组操作的过程中，如果一次仅取一个元素，尤其是逐个赋值时，会很耗费资源。所以最好使用 `findnz()` 函数先将稀疏矩阵先转换为 (I, J, V) 这种稠密向量结构，对其进行处理后，再重构稀疏矩阵对象。



提示 稀疏结构能够高效地存储稀疏的数据，而且在索引访问、赋值更新、迭代遍历等各方面都与普通的稠密数组相似。不过由于稀疏结构采用了压缩式的存取方式，所以最好遵循 CSC 结构的特点，避免耗时的操作。例如，应尽可能按列访问元素，以及避免插入新的元素等。

8.6 矢量化计算

在对数集进行操作时，经常遇到要对其内部元素实施同一种计算的情况。例如对数组中的每个元素求绝对值、平方；或以元素为参数调用某个函数，并将计算结果生成新的数组，等等。常规的做法是遍历数组，对每个元素单独地实现计算过程。但在密集数值计算的应用中，这种逐元 (elementwise) 计算的需求往往会频繁出现。如果使用了过多的循环遍历，不但逻辑变得冗杂繁复，而且会导致代码量变大、开发效率降低及维护成本增加。

为此，Julia 中提供了矢量化计算的机制，能够极为简洁地实现逐元计算，而且无须开发者自行实现循环遍历操作。下面将详细介绍几种实现方式。

8.6.1 map 函数

在 Julia 中，常规矢量化的方法是使用 `map()` 函数，其原型为：

```
map(f::Function, c...) -> collection
```

其中，参数 `f` 是一个函数对象，`c` 是一个可变参数的迭代数集组。调用时，`map()` 内部会自动对 `c` 中的每个元素执行 `f()` 函数，并将计算的结果组建为新的数集。

例如，我们通过该函数实现数组元素的四舍五入：

```
julia> map(round, [1.2, 3.5, 1.7])
3-element Array{Float64,1}:
 1.0
 4.0
 2.0
```

`map()` 函数也可以适用于元组、Range 等类型，例如：

```
julia> map(round, (1.2, 3.5, 1.7))
(1.0, 4.0, 2.0)

julia> map(round, 1.1:0.5:3.0)
4-element Array{Float64,1}:
 1.0
 2.0
```



```
2.0
3.0
```

而且在数集方面，还可以更为灵活地使用生成器。举例如下：

```
julia> map(tuple, (1/(i+j) for i=1:2, j=1:2), [1 3; 2 4])
2×2 Array{Tuple{Float64,Int64},2}:
 (0.5, 1)      (0.333333, 3)
 (0.333333, 2) (0.25, 4)
```

其中的 `tuple()` 函数会将 `map()` 的第二个生成器参数与第三个数组参数的对应元素组装为二元的元组结构。不过需要注意的是，其中的生成器需要使用圆括号界定，否则会出现语法错误。

一般情况下，`map()` 函数生成的结果数集结构总是会与输入的数集一致，即：输入数集是元组，结果类型便是元组；若是数组则结果数集也是数组，如此等等。

在函数对象参数方面，除了已有的函数能作为参数，也可以采用匿名函数。例如：

```
julia> map(x -> x * 2, [1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6
```

其中，`map()` 会将匿名函数 `x->x*2` 施用于数组 `[1, 2, 3]` 中的所有三个元素，然后放大两倍后的结果会存储在一个新的数组中。

对于复杂的函数实现，可以通过 `begin` 结构提供实现体。例如：

```
julia> map(x->begin
    if x < 0 && iseven(x)           # 小于0或偶数时返回0
        return 0
    elseif x == 0                   # 0时返回1
        return 1
    else                             # 其他情况返回原值
        return x
    end
end, [-2, 7, 8, 0])
4-element Array{Int64,1}:
 0
 7
 8
 1
```

该匿名函数对象略显复杂，虽然不会影响 `map()` 函数的正常使用，但会让 `map()` 的调用代码看起来极不清晰。可以考虑使用 `do` 代码块实现，这样，前例中的代码便可以修改为：

```
julia> map((-2, 7, 8, 0)) do x
    if x < 0 && iseven(x) # 小于0或偶数时返回0
        return 0
    elseif x == 0        # 0时返回1
        return 1
    else                  # 其他情况返回原值
        return x
    end
end
```

```

                                end
(0, 7, 8, 1)

```

另外，作为函数对象的一种，运算符也可以参与 `map()` 计算。例如：

```

julia> map(~, (1, 2, 3))
(-2, -3, -4)

```

其中的运算符 `~` 会将每个元素的二进制位进行取反处理。当然二元或多元运算符也是适用的，只需将 `map()` 传入更多的满足条件的数集作为操作数即可。例如：

```

julia> map(+, [1, 2, 3], [10, 20, 30])
3-element Array{Int64,1}:
 11
 22
 33

julia> map(+, [1, 2, 3], [10, 20, 30], [100, 200, 300])
3-element Array{Int64,1}:
111
222
333

```

可见，`map()` 函数支持可变参数，只需按照函数对象参数的需求提供对应的参数即可。

从使用方便及性能考虑，`map()` 还有一个就地修改版，即 `map!()` 函数，其原型为：

```

map!(f, destination, c...)

```

使用方式与上述的 `map()` 函数类似，不同的是 `destination` 参数是预先定义的数集，用于接收计算结果，并不参与计算过程。

在这种就地修改版中，因为 `destination` 用于存储计算结果，所以其维度或大小必须满足要求，一般不应小于输入数集序列 `c` 中的首个数集的大小。另外，因为 `destination` 会被 `map()` 改变，所以不能再是元组类型。

下面举例说该函数的使用方法：

```

julia> a = zeros(3);

julia> map!(x -> x * 2, a, [1, 2, 3]);

julia> a
3-element Array{Float64,1}:
 2.0
 4.0
 6.0

```

其中，将 `a` 初始化为 3 元素的零向量，经过 `map!()` 处理后便存储了放大两倍后的新元素。

再看另外一个例子：

```

julia> x = [1, 2, 3];

julia> map!(+, x, [10, 20, 30], [100, 200, 300])
3-element Array{Int64,1}:
110
220

```



```

330

julia> x
3-element Array{Int64,1}:
 110
 220
 330

```

其中，除了接收结果的数集 x 外，仅提供了两个额外数集。函数调用顺利执行后，逐元相加的结果存储在了 x 中，替换了 x 原来的值。

8.6.2 广播

Julia 还提供了一个广播函数，可用于矢量化操作，其原型为：

```
broadcast(f::Function, As...)
```

可见其与 `map()` 函数非常相似，在调用方式上也相同，例如：

```
julia> broadcast(+, (1,2,3), (2,3,4))
(3, 5, 7)
```

```
julia> broadcast(x->x*2, (2,3,4))
(4, 6, 8)
```

但在数集与其他非数集类型的混合计算中，广播函数与 `map()` 的表现会有所不同，例如：

```
julia> broadcast(+, (1,2,3), 2)
(3, 4, 5)
```

```
julia> map(+, (1,2,3), 2)
1-element Array{Int64,1}:
 3
```

其中，希望数集的每个元素均加 2，`broadcast()` 能够获得正确结果，而 `map()` 结果并不符合预期。再看下面的混合乘法的例子：

```
julia> broadcast(*, [1,2,3], 2)
3-element Array{Int64,1}:
 2
 4
 6
```

```
julia> map(*, [1,2,3], 2)
1-element Array{Int64,1}:
 2
```

其中，`broadcast()` 符合预期地逐个元素乘以 2，但 `map()` 的如前，未得到预期结果。

此外，在基于 `broadcast()` 函数实现矢量化混合计算时，对参数的顺序并没有严格的要求。例如：

```
julia> broadcast(*, 2, (1,2,3))
(2, 4, 6)
```

```
julia> broadcast(*, 2, (1,2,3), 3)
(6, 12, 18)
```

其中，标量可以与其中的数集处于参数表中的任意位置。而且，`broadcast()` 函数还支持维度不同的数集参与计算，例如：

```
julia> a = [1 2 3]
1×3 Array{Int64,2}:
 1  2  3

julia> b = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> broadcast(+, a, b)
3×3 Array{Int64,2}:
 2  3  4
 3  4  5
 4  5  6

julia> map(+, a, b)
3-element Array{Int64,1}:
 2
 4
 6
```

由此可见，`broadcast()` 与 `map()` 有显著的不同，能够更好地适应各种矢量化计算的应用场景，支持元组、数组、标量以及引用（Ref）等各种类型。默认情况下，数值类型、字符串、Symbol 类型、类型（Types）、函数以及一些单例对象（例如 missing 与 nothing）会被处理为标量，其他输入参数对象都会被进行矢量化处理，实施逐元计算。

在这些类型混合计算的时候，`broadcast()` 能够较好地甄别矢量化的计算需求，给出符合预期的结果。其生成结果的矢量化规则为：

- ❑ 如果参数均是标量或空维的数组，返回标量。
- ❑ 如果参数至少有一个是元组，而其他的是标量或空维的数组，则返回元组类型。
- ❑ 其他的类型组合一般均返回数组类型；若存在自定义的类型，则依据实现方法或提升规则返回恰当的类型。

8.6.3 点操作

对数组的逐元操作用途非常广泛，而且频度很高，为此 Julia 提供了一种“点操作符”，能够很方便地将已有的函数变成矢量化版本，实现逐元操作。

例如，定义一个函数 `foo()`，代码如下：

```
julia> foo = x->x^3
(::#1) (generic function with 1 method)
```

显然其不支持数组的操作，即：

```
julia> a = [ 2 3 4 5 ]
1×4 Array{Int64,2}:
```

```
2 3 4 5
```

```
julia> foo(a)
ERROR: DimensionMismatch("matrix A has dimensions (1,4), matrix B has dimensions (1,4)")
```

因为数组的幂操作本身也是无意义的。

但如果在函数名之后参数表之前加上一个英文句号（点操作符），便能够将函数转为矢量化版本，再对其调用时便可实现对数组的逐元操作，即：

```
julia> foo.(a)          # 函数名之后有一个点
1×4 Array{Int64,2}:
 8 27 64 125
```

而且，点操作不但适用于自定义的函数，同样可用于各种内置函数，例如：

```
julia> sin.(a)
1×4 Array{Float64,2}:
 0.909297 0.14112 -0.756802 -0.958924
```

再例如：

```
julia> b = [3 2 10 1]
1×4 Array{Int64,2}:
 3 2 10 1
```

```
julia> min.(a, b)
1×4 Array{Int64,2}:
 2 2 4 1
```

其中 `min.(a, b)` 会将数组 `a` 和 `b` 的对应元素进行比较，并返回每两个中的最小一个，生成新的结果数组。

不仅如此，矢量化作用的数集并不限于数组，也适用于其他类型的数集，例如：

```
julia> c = Tuple(a)
(2, 3, 4, 5)
```

```
julia> d = Tuple(b)
(3, 2, 10, 1)
```

```
julia> max.(c, d)
(3, 3, 10, 5)
```

```
julia> e = Set(a)
Set{Int64}([4, 2, 3, 5])
```

```
julia> f = Set(b)
Set{Int64}([10, 2, 3, 1])
```

```
julia> min.(e, f)
4-element Array{Int64,1}:
 4
 2
 3
 1
```

其中，`Set` 数集是无序的，一旦构建其顺序也是固定的，所以对其操作的矢量化 `min()` 函

数也可以正常运行。输入 Set 数集时，不过返回的类型不再像元组或数组那样与输入数集类型一致，而变成变成了数组。

更为方便地是，点操作符也可以对运算符转换，不过要留意的是，运算符的矢量化方式略有不同：如果在名字后附加点操作符，会报错：

```
julia> (1, 2, 3) *. 2
ERROR: syntax: invalid identifier name "."
```

实际上，此时进行矢量的正确方式是将点操作符加在运算符之前，即：

```
julia> (1, 2, 3) .* 2
(2, 4, 6)
```

该操作顺利地将数集的元素放大了两倍。实际上，此时的 `a .* b` 相当于 `(*) . (a, b)`，在内部实现原理上与 `broadcast(*, a, b)` 是一致的。

再例如，`[1, 2, 3]^3` 是不允许的，因为不可能对数组的整体进行幂运算，但 `[1, 2, 3].^3` 却自动将 `^` 操作矢量化，将幂运算用于数组的元素，相当于 `[1^3, 2^3, 3^3]`，即：

```
julia> [1,2,3] .^ 3
3-element Array{Int64,1}:
 1
 8
27
```

```
julia> broadcast(^, [1,2,3],3)
3-element Array{Int64,1}:
 1
 8
27
```

正由于 `broadcast()` 与点操作符的相通性，所以点操作符构造的矢量化版函数或运算符也支持数集与标量等类型的混合计算，而且也同样支持不同维度或阶数的数组。不过显而易见的是，点操作更为方便直观。

如果表达式中连续多次地出现点操作，Julia 内部会自动将内部的逐元操作进行融合，避免多次的循环。例如，`sin.(cos.(A))` 在执行内部，不会先对 `A` 生成余弦结果的数集，再对该数集求正弦结果；而是转为一次循环操作，对数集 `A` 逐元地求余弦再求正弦后才会生成新的数集。

在一些情况下，恰当地使用点操作，能够大幅度地减少计算复杂度。例如：

```
julia> ((1:10000) .+ 20) .* 7
147:7:70140
```

Julia 能够自动识别出其中序列 `1:10000` 为 `Range` 类型，在计算时根本不会真的分配出 10000 个元素，也不需要执行 10000 次计算，实际上只会基于该 `Range` 对象进行操作，生成一个新的 `Range` 对象，从而能够将 $O(N)$ 的复杂度降低到 $O(1)$ 的复杂度。

另外，点操作也适用于用户自定义的运算符。例如，如果定义 $A \otimes B = \text{kron}(A, B)$ 以代替中缀形式 `A ⊗ B` 计算克罗内克积，之后若要计算 `[A ⊗ C, B ⊗ D]`，则用 `[A,B] . ⊗ [C,D]` 即可，不需要额外的代码编写工作。

8.6.4 数组运算符

Julia 也为数组提供了适用的运算符，如表 8-4 所示。

表 8-4 数组运算符

分 类	符 号
一元算术运算符	- +
二元算术运算符	- + * / \ ^
比较	== != ≈ ≠

其中的一些运算符是支持矢量化运算的，例如：

```
julia> a = [2 3 4];
```

```
julia> b = -a
1×3 Array{Int64,2}:
-2 -3 -4
```

```
julia> a * 10
1×3 Array{Int64,2}:
20 30 40
```

```
julia> a / 4.0
1×3 Array{Float64,2}:
0.5 0.75 1.0
```

在 v1.0 版之前，加减法也是直接支持矢量化计算的，但因为冲突问题已经被舍弃，所以需要采用点操作实现逐元计算，例如：

```
julia> a .- 10
1×3 Array{Int64,2}:
-8 -7 -6
```

```
julia> a .+ 10
1×3 Array{Int64,2}:
12 13 14
```

同样，对于右操作数是数组的情况，也需点操作符进行矢量化转换，例如：

```
julia> 4. \ a
1×3 Array{Float64,2}:
0.5 0.75 1.0
```

另外，运算符 == 等在对两个数组进行对比时，也并不是矢量化的，实际比较的是所有元素，只有当所有元素均相等才会返回 true 否则返回 false，例如：

```
julia> [1,2,3] == [1,2,3]
true
```

```
julia> [1,2,3] == [1,2,5]
false
```

所以要得到逐元比较的结果，需将这类运算符转为矢量化版本，例如：

```
julia> [1 2 3] .== [1 2 3]
1×3 BitArray{2}:
 true  true  true
```

此时得到的便是逐个元素的对比结果。

8.7 排序

对于数组类的序列性容器，Julia 提供了可扩展的、灵活的 API 用于对其进行排序。不但支持多种排序算法，还支持在排序时预先对元素进行必要的转换，同时还直接提供了就地修改版本，这样我们在开发时便有了多种选择。

所谓排序算法 (Base.Sort.Algorithm) 是指比较序列中元素大小的逻辑，以及对位置进行调整并生成有序数组的流程方案。Julia 中提供了三种常见算法，包括快速排序、插入排序及归并排序。有关排序算法的详细介绍，读者可自行参阅数据结构等方面的资料，下面仅进行简单地介绍。

- ❑ 快速排序 (Base.Sort.QuickSort) 采用了就地修改的策略，复杂度为 $O(n \log n)$ ，速度较快，但不稳定（等值元素在序列中的顺序会变化），其在数值型序列的排序中是 Julia 默认的算法。
- ❑ 插入排序 (Base.Sort.InsertionSort) 是一种稳定的排序算法，其复杂度为 $O(n^2)$ ，当元素不多 (n 较小) 时，有着较高的性能，也是快速排序内部一个子算法。
- ❑ 归并排序 (Base.Sort.MergeSort) 并没有采用就地修改的方式，需要创建与待排序数组等大小的临时空间，其复杂度虽然与快速排序一样，但一般没有快速排序性能高，是 Julia 对非数值型等其他数据排序时的默认算法。

Julia 语言中基本排序方法的原型为：

```
sort(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false,
    order::Ordering=Forward)
sort!(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::Bool=false,
    order::Ordering=Forward)
```

其中，参数 v 是待排序的序列，而其他则是可选的键值参数。两个方法的不同之处在于是否采用就地修改，即排序的结果是否会覆盖原数组 v 变量的内容。

可选参数 alg 用于指定排序算法，不指定时内部会根据数据类型自动选择；可选参数 by 指定在排序前要对元素做何种处理；参数 lt 用于指定元素间比较时所采用的方法；而参数 rev 指定序列的排列方式，是降序还是升序，未指定时为升序。函数中的 alg 、 by 、 lt 及 rev 四个参数是相互独立的，可自由组合。当 by 和 lt 同时被设定时， lt 函数会施用在 by 函数调用后返回的结果上。

例如，我们需要对一个列向量按升序排序，如下：

```
julia> sort([2,3,1])
3-element Array{Int64,1}:
```

```
1
2
3
```

如果需要降序，则可以：

```
julia> sort([2,3,1], rev=true)
3-element Array{Int64,1}:
 3
 2
 1
```

或者以变换后的值进行排序：

```
julia> sort([2, -3, -1], by=abs, rev=true)
3-element Array{Int64,1}:
-3      # abs(-3) = 3
 2      # abs(2)  = 2
-1      # abs(-1) = 1
```

但转换的值仅用于排序，不会作为排序的结果。

对于序列中元素为复合结构时（例如元组、字典等），我们能够利用 `by` 参数依据复合结构的某个字段进行排序。例如：

```
julia> sort([(1, "b"), (2, "a")], by = x -> x[1]) # 按元组内部的第1个元素排序
2-element Array{Tuple{Int64,String},1}:
(1, "b")
(2, "a")
```

```
julia> sort([(1, "b"), (2, "a")], by = x -> x[2]) # 按元组内部的第2个元素排序
2-element Array{Tuple{Int64,String},1}:
(2, "a")
(1, "b")
```

```
julia> x = ["a"=>2, "c"=>1, "b"=>3] # 定义一个Pair类型的数组
3-element Array{Pair{String,Int64},1}:
"a" => 2
"c"  => 1
"b"  => 3
```

```
julia> sort(x, by=p->p.first) # 按Pair的键排序
3-element Array{Pair{String,Int64},1}:
"a" => 2
"b" => 3
"c" => 1
```

```
julia> sort(x, by=p->p[2]) # 按Pair的值排序
3-element Array{Pair{String,Int64},1}:
"c" => 1
"a" => 2
"b" => 3
```

上述的两个方法仅适用于列向量（一维数组），如果要对其他结构的高维数组排序，需要使用以下的方法原型：

```
sort(A; dims::Integer, alg::Algorithm=DEFAULT_UNSTABLE, lt=isless,
      by=identity, rev::Bool=false, order::Ordering=Forward)
```

其中, `dims` 用于指定需要排序的维度。例如:

```
julia> a = [13 2 3 12 9 20 1]
1×7 Array{Int64,2}:
 13  2  3 12  9 20  1

julia> sort(a, dims = 2)                                     # 对第2维(列)排序
1×7 Array{Int64,2}:
 1  2  3  9 12 13 20
```

其中, `a` 为行向量(一行的二维数组), 对其排序时需指定要排序的维度。

再例如, 对多行矩阵排序, 如下所示:

```
julia> b = [3 1 5; 2 10 7; 3 2 9]
3×3 Array{Int64,2}:
 3  1  5
 2 10  7
 3  2  9

julia> sort(b, dims=1, alg=MergeSort)                       # 采用归并排序
3×3 Array{Int64,2}:
 2  1  5
 3  2  7
 3 10  9
```

通过指定维度实现对矩阵的各列分别进行排序。

另外一个排序的场景, 即不需要有序的值序列, 而要获得序列值排序后的各元素的原下标值, 可使用以下函数:

```
sortperm(v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless,
         by=identity, rev::Bool=false, order::Ordering=Forward)
```

该方法默认采用归并排序算法, 当然也可通过 `alg` 指定其他算法。例如:

```
julia> c = [20, -30, -10]
3-element Array{Int64,1}:
 20
-30
-10

julia> p = sortperm(c, alg=QuickSort, by=abs, rev = true)    # 用快速排序方法对序列的绝对值排序
3-element Array{Int64,1}:
 2      # 对应-30
 1      # 对应20
 3      # 对应-10

julia> c[p]
3-element Array{Int64,1}:
-30
 20
-10                                     # 结果序列按绝对值从大到小排列
```

此外, 对于复合类型, 我们可以通过自定义的比较函数实现对 `sort` 方法的扩展。例如, 有一个 `Person` 类型, 其结构为:

```
julia> mutable struct Person
```



```

    age::Int64 # 年龄
    # 其他体重、身高、胖瘦等成员
end

```

为其定义比较函数，使得该复合类型有序：

```

julia> lessthan(a::Person, b::Person) = a.age < b.age
lessthan (generic function with 1 method)

```

此后，便可实现对 Person 类型数组的排序。例如：

```

julia> ps = [Person(21), Person(32), Person(20), Person(15)]
4-element Array{Person,1}:
 Person(21)
 Person(32)
 Person(20)
 Person(15)

julia> sort(ps, lt=lessthan)                                     # 按年龄升序排序
4-element Array{Person,1}:
 Person(15)
 Person(20)
 Person(21)
 Person(32)

```

当然，在排序之前，我们可以通过函数判断一个数组向量是否已经有序：

```

issorted(v, lt=isless, by=identity, rev::Bool=false, order::Ordering=Forward)

```

其中，参数 by、lt 与 rev 意义同上，在此处用于限定有序性检测的条件与规则。例如：

```

julia> issorted([1, 2, 3])
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[1])           # 是否按照Tuple的第1个元素排序
true

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2])           # 是否按照Tuple的第2个元素排序
false

julia> issorted([(1, "b"), (2, "a")], by = x -> x[2], rev=true)
true

```

8.8 查找

1. 存在判断

在查找某个值在数集中的位置之前，可以先用 in() 判断其是否在该数集中存在。该函数的原型为：

```

in(x, v)

```

如果 x 在 v 中存在，返回 true，否则返回 false 值。例如：

```

julia> s = [1 2 3 4];
julia> in(1, s)
true

```

```
true
```

```
julia> in(5, s)
false
```

该函数在实际应用中需要留意参数的顺序：待搜索的序列是第二个参数，而待查找的值是第一个参数。因为 `in()` 并没有限定参数的类型，所以如果传参反了仍可以正常工作，但得到的结果肯定不是预期的。

该函数在使用中，提供了一种更为直观的用法，即 `x in v`，例如：

```
julia> 1 in a
true
```

```
julia> 5 in a
false
```

或者：

```
julia> 1 in [1, missing]
true
```

```
julia> missing in Set([1, 2])
false
```

需要注意其中 `missing` 在不同情况下的表现。与上例中不同的是下述的情况：

```
julia> missing in [1, 2]
missing
```

```
julia> 1 in [2, missing]
missing
```

虽然 `missing` 不在 `[1, 2]` 中，但 `in()` 并没有返回 `false`，而是返回了 `missing`，这是因为 `missing` 是不确定的数据，所以对其判断同样会返回不确定的结果。而在 `[2, missing]` 中，字面上 `1` 是不存在的，但因为数集中有 `missing`，而我们不确定这个 `missing` 是否事实上为 `1` 值，所以只能返回 `missing`。

另外，`in()` 函数支持自定义的类型，不过需要该类型已经定义了准确的运算符 `==` 操作，因为在其进行是否存在的判断时，检测结果依赖于该运算符。

2. 有序定位

若要查找某个给定值在有序序列中的位置，可以通过以下三个方法：

```
searchsortedfirst(v, x, [by=<transform>,] [lt=<comparison>,] [rev=false])
searchsortedlast(v, x, [by=<transform>,] [lt=<comparison>,] [rev=false])
searchsorted(v, x, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

即在 `v` 中搜索 `x` 的位置，其中参数 `by`、`lt` 及 `rev` 的意义同上。第一个方法会返回 `x` 在 `v` 中第一次出现的索引值；第二个方法会返回 `x` 在 `v` 中最后一次出现的索引值。因为 `v` 是有序的，所以相同的值多次出现时会连续地排列在一起。上述的第三个方法会以范围表达式的方式给出搜索到的某个值 `x` 的起止范围。

例如，一个有序一维数组 `a`，其中有 7 个元素并已按升序排列，分别调用上述三个函

数得到的结果为：

```
julia> a = [1;2;2;3;4;4;5]
7-element Array{Int64,1}:
 1
 2      # 索引值为2
 2      # 索引值为3
 3
 4
 4
 5

julia> searchsortedfirst(a, 2)
2

julia> searchsortedlast(a, 2)
3

julia> searchsorted(a, 2)
2:3

julia> typeof(ans)
UnitRange{Int64}
```

其中，`searchsorted()` 返回的结果为 `UnitRange{Int64}` 类型，记录了被搜索值所在的范围。



提示 在使用上述三个方法前，最好使用 `issorted()` 方法预先判断搜索的序列是否已经是有序的，否则得到的结果可能不是预期的。

3. 极值定位

极值包括最大值与最小值，可以通过以下函数进行定位：

```
findmax(v) -> (maxval, index)
findmin(v) -> (minval, index)
```

这两个方法在将找到的极值位置 `index` 返回的同时，还会直接返回该极值的具体取值。例如：

```
julia> findmax(s)
(4, 6)      # 最大值为4，索引值为6

julia> findmin(s)
(1, 1)      # 最小值为1，首个索引值为1
```

可见，当极值多次存在时，只会记录首次出现的位置。

另外，对于多维数组，返回的位置是一个 `CartesianIndex` 对象，记录着极值所处笛卡尔坐标。例如：

```
julia> a = [10 20; 30 40; 50 60]
3×2 Array{Int64,2}:
 10  20
 30  40
 50  60
```

50 60

```
julia> findmax(a)
(60, CartesianIndex(3, 2))          # 最大值为60, 笛卡尔坐标为(3,2)
```

```
julia> findmin(a)
(10, CartesianIndex(1, 1))          # 最小值为10, 笛卡尔坐标为(1,1)
```

若要单独地查找各行或各列中的极值, 可以使用此两个方法的另外两个原型:

```
findmax(v; dims) -> (maxval, index)
findmin(v; dims) -> (minval, index)
```

其中, `dims` 用于限定搜索的维度, 可以是单个维度或多个维度组合。例如:

```
julia> findmax(a, dims=1)          # 在行序中搜索最大值
([50 60], CartesianIndex{2}[CartesianIndex(3, 1) CartesianIndex(3, 2)])
```

```
julia> findmin(a, dims=2)          # 在列序中搜索最小值
([10; 30; 50], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 1);
CartesianIndex(3, 1)])
```

```
julia> findmax(a, dims=(1,2))      # 获得在行序及列序中都是最大值的信息
([60], CartesianIndex{2}[CartesianIndex(3, 2)])
```

在使用这两个函数时, 待搜索数集不能为空, 即至少有一个元素, 否则会报错。还有另外一个需要注意的是, 如果数据中存 `NaN` 类型的值, 无论找最小值还是最大值, 都会返回该值。例如:

```
julia> findmax([1,7,7,NaN])
(NaN, 4)
```

```
julia> findmin([7,1,1,NaN])
(NaN, 4)
```

我们可以使用 `filter(!isnan, v)` 先将 `v` 中的所有 `NaN` 元素过滤掉, 再调用这两个函数求取极值, 即:

```
julia> findmax(filter(!isnan, [1,7,7,NaN]))
(7.0, 2)
```

```
julia> findmin(filter(!isnan, [1,7,7,NaN]))
(1.0, 1)
```

至于结果为何会由整型数值变成浮点值, 有兴趣的读者可以研究一下。

4. 自定义定位

Julia 还有一个强大的功能——`findall()` 提供了自定义的接口:

```
findall(f::Function, v)
```

其中, 函数 `f` 提供了搜索条件, 在元素是否满足时返回对应的 `Bool` 值。该函数如果找不到结果, 会返回空集。

该函数一个典型的应用便是找到某个具体的值。如果序列是无序的, 用法为:

```
findall(in(x), v)
```


该方法不关心待搜索数集是否有序，而且支持多位置搜索，能够返回一个数组，记录着被搜索值 x 在 v 中出现的所有索引位置。该方法的第一个参数相当于匿名函数 $x \rightarrow x \text{ in } v$ ，使用方法示例如下：

```
julia> s = (1,1,2,3,2,4,1);
```

```
julia> findall(in(1), s)
3-element Array{Int64,1}:
 1
 2
 7
```

```
julia> findall(in(1), s)
2-element Array{Int64,1}:
 3
 5
```

可见，其中被搜索值 1 出现了三次而且并不连续，返回的数组正记录了三次的位置。

正是由于 `findall()` 函数首个参数的提供了搜索条件的接口，所以我们能够非常方便地在被搜索数据中找到各种想要的数。例如，在数集中搜索所有偶数的位置，可以如下实现：

```
julia> findall(iseven, 1:4)
2-element Array{Int64,1}:
 2
 4

julia> findall(x->x%2==0, c)
3-element Array{CartesianIndex{2},1}:
 CartesianIndex(1, 2)
 CartesianIndex(2, 2)
 CartesianIndex(1, 3)
```

再例如，找到所有非零元素的位置：

```
julia> findall(!iszero, c)
4-element Array{CartesianIndex{2},1}: ## 非零元素笛卡尔坐标列表
 CartesianIndex(1, 1)
 CartesianIndex(2, 1)
 CartesianIndex(1, 3)
 CartesianIndex(2, 3)
```

该函数一个强大的特性是能够针对不同的数据类型返回不同的索引结构。例如，如果被搜索数集是字典类型，会自动在值域内搜索，并返回键的集合：

```
julia> d = Dict{"A" => 10, "B" => -1, "C" => 0}
Dict{String,Int64} with 3 entries:
  "B" => -1
  "A" => 10
  "C" => 0
```

```
julia> findall(x -> x >= 0, d)
2-element Array{String,1}:
 "A"
 "C"
```

另外，由于该方法中的函数对象是第一个参数，所以可以使用 `do` 结构。例如：

```
julia> findall(c) do x
    iszero(x)
end
2-element Array{CartesianIndex{2},1}:
 CartesianIndex{1, 2}
 CartesianIndex{2, 2}
```

在查找方面，除了以上介绍的这些之外，还有更多灵活的函数方法，有兴趣的读者可自行学习。

8.9 missing 作为元素

数据缺失是经常会遇到的，所幸的是，在用 Julia 创建数组对象时，可将 `missing` 作为其中的元素。例如：

```
julia> [1, missing]
2-element Array{Union{Missing, Int64},1}:
 1
 missing
```

此时的元素类型会变成 `Union{Missing, T}`，其中的 `T` 则是非缺失值的类型。此时，内部会使用一个 `Array{UInt8}` 类型的对象记录元素是否为 `Missing` 或是 `T`，但存取性能与普通的数组类型 `Array{T}` 是相当的。

当然，在构造数组对象时，可以显式地使用带有 `Missing` 类参的构造方法，例如：

```
julia> Array{Union{Missing, String}}(missing, 2, 3)
2×3 Array{Union{Missing, String},2}:
 missing missing missing
 missing missing missing
```

对于这种 `Union{Missing, T}` 类型的数组，可以转换到普通的 `Array{T}` 类型数组，例如：

```
julia> x = Union{Missing, String}["a", "b"]
2-element Array{Union{Missing, String},1}:
 "a"
 "b"

julia> convert(Array{String}, x)
2-element Array{String,1}:
 "a"
 "b"
```

如果 `Union{Missing, T}` 类型的数组中确实存在了 `Missing` 类型的值，会抛出 `MethodError` 错误：

```
julia> y = Union{Missing, String}[missing, "b"]
2-element Array{Union{Missing, String},1}:
 missing
 "b"

julia> convert(Array{String}, y)
```

```
ERROR: MethodError: Cannot `convert` an object of type Missing to an object of type String
```

如 6.10.1 节所述, 缺失性会在数学运算、聚合函数等操作中进行传播, 导致无法得到正常的结果。这种情况对于数组而言同样是存在的, 例如:

```
julia> sum([1, missing])
missing
```

如果处理数据时我们能够确定那些缺失值毫无意义, 可以抛弃, 便可调用 `skipmissing()` 函数过滤掉缺失值, 再进行后续的处理, 例如:

```
julia> sum(skipmissing([1, missing]))
1
```

其中, `skipmissing()` 函数会返回 `Base.SkipMissing` 类型的结构, 这是一种迭代器, 能够高效地跳过所有的 `missing` 值, 所以适用于那些支持迭代器作为参数的任意函数。再例如:

```
julia> maximum(skipmissing([3, missing, 2, 1]))
3
```

```
julia> mean(skipmissing([3, missing, 2, 1]))
2.0
```

```
julia> mapreduce(sqrt, +, skipmissing([3, missing, 2, 1]))
4.146264369941973
```

当然, 也可以使用 `collect()` 函数将 `skipmissing()` 的迭代结构转为元素均为有效值的数组, 例如:

```
julia> collect(skipmissing([3, missing, 2, 1]))
3-element Array{Int64,1}:
 3
 2
 1
```

在对数组本身进行逻辑判断时 (是否同位置的元素均相等), `missing` 值也会影响正常结果的输出, 例如:

```
julia> [1, missing] == [2, missing]
false
```

```
julia> [1, missing] == [1, missing]
missing
```

```
julia> [1, 2, missing] == [1, missing, 2]
missing
```

上例中, 第一个语句因为同位置的非 `missing` 元素就不相同, 所以无论其他元素如何都能够判断这两个数组是不相等的; 但第二个语句中, 虽然元素的内容都“相同”, 但 `missing` 是无法确定的值, 所以两个 `missing` 之间是否相等的判断是没有依据的, 只能输出判定结果也是 `missing` 值, 无法得到 `true` 或 `false` 的结论。对于第三个语句, 对

应位置有的为 missing 值，有的没有 missing 值，这同样无法让我们知晓同位置的元素实际是否真的相等，所以也只能返回 missing 的判断。

不过，在对待这些 missing 值时，如果能够简单粗暴地认为它们就是同一个“东西”，不用进行区分，那么可以调用 `isequal()` 函数进行相等性判断，例如：

```
julia> isequal([1, missing], [1, missing])
true
```

此时，missing 值会被当成一个普通数据，而内容就是字面值 missing，所以上面的语句不再像上面那样“无可奈何”地返回 missing 这种结论，而是能够给出确切的 true 或 false。例如：

```
julia> isequal([1, 2, missing], [1, missing, 2])
false
```

不过，对于数据分析或挖掘的实际问题中，missing 的处理往往不会如此简单。如果可以的话，最好以均值、默认值等方式对 missing 值进行替代，或者使用预测、相关性分析等手段对这些缺失值进行填充。至于哪种方式最好，需要根据实际问题进行选择。

8.10 线性代数中的矩阵处理

从前面的介绍中可以看出，Julia 的数组结构是非常强大的，适用性极强。不但如此，Julia 还提供了大量线性代数计算方式。这里专门介绍矩阵的处理方式。

8.10.1 矩阵操作

Julia 在内置的 LinearAlgebra 模块中，针对矩阵提供了常见的线性代数运算，例如迹 (trace)、行列式值 (det)、逆 (inverse) 等。

假设矩阵 A，如下所示：

```
julia> A = rand(3,3)
3×3 Array{Float64,2}:
 0.789521  0.513853  0.366957
 0.422566  0.532127  0.733008
 0.470651  0.802462  0.267549
```

可分别调用 `tr()`、`det()` 和 `inv()` 函数，求取其迹、行列式值以及逆矩阵，即：

```
julia> using LinearAlgebra
```

```
julia> tr(A)
1.589195866245045
```

```
julia> det(A)
-0.20029116218223145
```

```
julia> inv(A)
3×3 Array{Float64,2}:
 2.22597  -0.783799  -0.905637
```



```
-1.15798 -0.192353 2.11523
-0.442591 1.95573 -1.01347
```

另外，也可以在矩阵后附加单引号 ' 计算矩阵的转置 (transpose)，即：

```
julia> A'
3×3 Array{Float64,2}:
 0.789521  0.422566  0.470651
 0.513853  0.532127  0.802462
 0.366957  0.733008  0.267549
```

也能够很方便地求矩阵的特征值 (eigenvalues) 与特征向量 (eigenvectors)，只需调用 `eigvals()` 和 `eigvecs()` 函数即可，例如：

```
julia> eigvals(A)
3-element Array{Float64,1}:
 1.63931
 0.325385
-0.375495

julia> eigvecs(A)
3×3 Array{Float64,2}:
 0.593206  0.805509  0.0398617
 0.589431 -0.495827 -0.638968
 0.54834 -0.324516  0.7682
```

也能够支持复数，例如：

```
julia> B = [1.5 2 -4; 3 -1 -6; -10 2.3 4]
3×3 Array{Float64,2}:
 1.5  2.0 -4.0
 3.0 -1.0 -6.0
-10.0 2.3  4.0

julia> eigvals(B)
3-element Array{Complex{Float64},1}:
 9.31908+0.0im
-2.40954+2.72095im
-2.40954-2.72095im

julia> eigvecs(B)
3×3 Array{Complex{Float64},2}:
-0.488645+0.0im  0.182546-0.39813im  0.182546+0.39813im
-0.540358+0.0im  0.692926+0.0im  0.692926-0.0im
 0.68501+0.0im  0.254058-0.513301im  0.254058+0.513301im
```

除此之外，更多的线性代数计算可参考手册。

8.10.2 特殊矩阵

在线性代数及常见的矩阵分解中，会出现不少对称或其他特殊结构的矩阵。为了方便这些特殊矩阵的常见操作，Julia 特别定义了一些特殊矩阵类型，如表 8-5 所示。

表 8-5 特殊矩阵类型

类 型 名	描 述
Symmetric	对称矩阵
Diagonal	单对角矩阵
Bidiagonal	双对角矩阵
Tridiagonal	三对角矩阵
SymTridiagonal	对称三对角矩阵
UpperTriangular	上三角矩阵
LowerTriangular	下三角矩阵
UniformScaling	统一缩放运算符
Hermitian	厄米特（厄米或埃尔米特）矩阵

假设有 A 矩阵，如下所示：

```
julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9
```

可以该矩阵为基础，可生成各种特殊矩阵的视图，例如：

```
julia> using LinearAlgebra

julia> a1 = Symmetric(A,:U)          # 以上三角部分构造对称矩阵，Symbol类型:U表示上三角方式
3×3 Symmetric{Int64,Array{Int64,2}}:
 1  2  3
 2  5  6
 3  6  9

julia> a2 = Bidiagonal(A, :L)        # 单对角及下次角构造双对角矩阵，:L表示获得下三角方式
3×3 Bidiagonal{Int64,Array{Int64,1}}:
 1  .  .
 4  5  .
 .  8  9

julia> a3 = Tridiagonal(A)
3×3 Tridiagonal{Int64,Array{Int64,1}}:
 1  2  .
 4  5  6
 .  8  9

julia> a4 = UpperTriangular(A)
3×3 UpperTriangular{Int64,Array{Int64,2}}:
 1  2  3
 .  5  6
 .  .  9

julia> a5 = Hermitian(A, :L)
3×3 Hermitian{Int64,Array{Int64,2}}:
```

```

1 4 7
4 5 8
7 8 9

```

另外, `Diagonal` 也可以将行向量或列向量作对角元素构造出方阵, 例如:

```

julia> V = [1; 2]
2-element Array{Int64,1}:
 1
 2

julia> d = Diagonal(V)
2×2 Diagonal{Int64,Array{Int64,1}}:
 1 .
 . 2

```

类似地, 可以由向量 `dv` 以及 `ev` 构造 `SymTridiagonal` 对象, 例如:

```

julia> dv = [1; 2; 3; 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7; 8; 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> st = SymTridiagonal(dv, ev)
4×4 SymTridiagonal{Int64,Array{Int64,1}}:
 1 7 . .
 7 2 8 .
 . 8 3 9
 . . 9 4

```

其中, `dv` 向量元素会成为结果方阵的主对角元素, 而 `ev` 则会成为次对角元素。

需要注意的是, 上述构造的并非新的对象, 而是原数组或向量 (`A`、`dv` 或 `ev`) 的视图, 所以对这些特殊矩阵中元素的修改, 会传递到原对象中。例如:

```

julia> a4[2,3] = 20;    # 修改上三角的一个元素

julia> a4
3×3 UpperTriangular{Int64,Array{Int64,2}}:
 1 2 3
 . 5 20
 . . 9
      # 被修改的位置

julia> A
3×3 Array{Int64,2}:
 1 2 3
 4 5 20
 7 8 9
      # 影响到了源数组

```

再例如:

```
julia> ev[2] = 100;

julia> st[3,2] == st[2,3] == ev[2] == 100
true
```

其中, `ev` 发生变化后, 以其构造的对称三对角阵中相应位置的元素值也发生了变化。

对于上述的特殊矩阵, 同样可以进行多种计算操作, 而且其中不少操作已由 Julia 基于 Lapack^①进行了优化。具体可见表 8-6。

表 8-6 特殊矩阵 Lapack 优化情况

类 型 名	+	-	*	\	其 他 函 数
Hermitian				MV	inv(),sqrt(),exp()
UpperTriangular			MV	MV	inv(),det()
LowerTriangular			MV	MV	inv(),det()
SymTridiagonal	M	M	MS	MV	eigmax(),eigmin()
Tridiagonal	M	M	MS	MV	
Bidiagonal	M	M	MS	MV	
Diagonal	M	M	MV	MV	inv(),det(),logdet(),/()
UniformScaling	M	M	MVS	MVS	/()

注: M 表明该运算符在矩阵对矩阵的情况下已优化, V 表明在矩阵对向量的情况下已优化, S 表明在矩阵对标量的情况下已优化。

8.10.3 矩阵分解

在线性代数中, 将矩阵分解 (Matrix Factorization 或 Matrix Decomposition) 为一些具有特殊属性的其他矩阵的组合, 可实现各种空间转换, 并会为后续的处理带来极大的便利。有多种矩阵分解方式, 包括 LU 分解 (又称为三角分解)、QR 分解及奇异值分解 (Singular Value Decomposition, SVD) 等, 常见的分解在 Julia 中均有实现。

我们首先简单回顾矩阵分解的基本定义:

- LU 分解。数值分析中, 在求解多元一次线性方程组时, 高斯消元法通过初等行变换逐步剥离出每个变量的求解式, 在这个过程中, 当系数矩阵 A 的所有顺序主子式都不为 0 时, A 会转变成矩阵 L 与 U 的乘积, 其中 L 和 U 分别是单位下三角矩阵与上三角矩阵。
- Cholesky 分解。当系数矩阵 A 是实对称正定矩阵时, A 矩阵可以表达为一个下三角矩阵 L 与其转置的乘积, 此时 L 的所有对角元素都是大于零的值。这种情况下, LU 分解就变成了 Cholesky 分解, 也被称为平方根法。

① LAPACK 是解决线性代数问题极负盛名的开源包, 由美国国家科学基金等资助开发, 可求解科学与工程计算中最常见的数值计算问题, 如线性方程组求解、最小二乘法、特征值和奇异值求解等。官网为 <http://www.netlib.org/lapack/>。

- ❑ QR 分解。将一个非奇异矩阵 A 分解为正交矩阵（或酉矩阵） Q 与非奇异上三角矩阵 R 的乘积，即为 QR 分解。该分解法是求解矩阵特征值、特征向量及矩阵的逆等问题的有效并常用的方法。
 - ❑ SVD 分解。这是一种将矩阵 A 分解为三个矩阵乘积的一种分解方法，表达为 $A = Q \Sigma Q^{-1}$ ，其中 Q 由矩阵 A 的特征向量组成，而 Σ 则是对角线为 A 特征值的对角阵。该分解时正规矩阵酉对角化的推广，是谱分析理论推广到任意矩阵时常用的方法，在信号处理、统计学、模式识别（Pattern Recognition）中有着重要的应用。SVD 可用于求解任意矩阵的伪逆矩阵，并能够用于进行主成分分析（Principal Component Analysis, PCA），从而进行隐含模式的提取，或数据压缩降维等。
- 在 Julia 中，具体分解及变体的实现，可见表 8-7。

表 8-7 矩阵分解类型

分解类型	描述
Cholesky	Cholesky 分解
CholeskyPivoted	Pivoted Cholesky 分解
LU	LU 分解
LUTridiagonal	三角矩阵的 LU 分解
UmfpackLU	系数矩阵的 LU 分解
QR	QR 分解
QRCompactWY	紧致 QR 分解
QRPivoted	PivotedQR 分解
Hessenberg	Hessenberg 分解
Eigen	频谱分解
SVD	奇异值分解
GeneralizedSVD	正规化的 SVD

下面给出一些使用的例子：

```
julia> using LinearAlgebra

julia> A = [1. 2.; 2. 50.]
2×2 Array{Float64,2}:
 1.0  2.0
 2.0 50.0

julia> B = [1.0 2.0; 3.5 50.0]
2×2 Array{Float64,2}:
 1.0  2.0
 3.5 50.0

julia> C = cholesky(A) # Cholesky分解，要求A必须是symmetric/Hermitian矩阵
Cholesky{Float64,Array{Float64,2}}
U factor:
```

```
2×2 UpperTriangular{Float64,Array{Float64,2}}:
 1.0  2.0
 .    6.78233
```

```
julia> C.U
2×2 UpperTriangular{Float64,Array{Float64,2}}:
 1.0  2.0
 .    6.78233
```

```
julia> C.L
2×2 LowerTriangular{Float64,Array{Float64,2}}:
 1.0  .
 2.0  6.78233
```

```
julia> C.L * C.U == A      # Cholesky分解还原
true
```

```
julia> Q, R = qr(B)      # QR分解
LinearAlgebra.QRCompactWY{Float64,Array{Float64,2}}
Q factor:
2×2 LinearAlgebra.QRCompactWYQ{Float64,Array{Float64,2}}:
 -0.274721 -0.961524
 -0.961524  0.274721
R factor:
2×2 Array{Float64,2}:
 -3.64005 -48.6256
  0.0     11.813
```

```
julia> Q * R == B      # QR还原
true
```

```
julia> L,U,p = lu(B)      # LU分解
LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0  0.0
 0.285714  1.0
U factor:
2×2 Array{Float64,2}:
 3.5  50.0
 0.0 -12.2857
```

```
julia> L
2×2 Array{Float64,2}:
 1.0  0.0
 0.285714  1.0
```

```
julia> U
2×2 Array{Float64,2}:
 3.5  50.0
 0.0 -12.2857
```

```
julia> p
2-element Array{Int64,1}:
 2
 1
```

```

julia> B[p,:] == L * U          # LU还原
true

julia> U, S, V = svd(B)          # SVD分解
SVD{Float64,Float64,Array{Float64,2}}([0.0411752 0.999152; 0.999152 -0.0411752],
 [50.1649, 0.857173], [0.0705316 0.99751; 0.99751 -0.0705316])

julia> U
2×2 Array{Float64,2}:
 0.0411752  0.999152
 0.999152  -0.0411752

julia> S
2-element Array{Float64,1}:
 50.16488068168744
 0.8571733733973987

julia> V
2×2 Adjoint{Float64,Array{Float64,2}}:
 0.0705316  0.99751
 0.99751   -0.0705316

julia> U * diagm(0=>S)*V == B   # SVD还原
true

# 或者
julia> U * Diagonal(S) * V == B
true

```

其中, `diagm()` 函数的原型为 `diagm(kv::Pair{<:Integer,<:AbstractVector})`, 用于创建一个方阵, 其第 `k.first` 个对角内容为向量 `k.second` 中的元素。例如:

```

julia> diagm(1=>[1,2,3])        # 正数1表示上三角的第1个次对角线, 反之负数表示下三角的位置
4×4 Array{Int64,2}:
 0  1  0  0
 0  0  2  0
 0  0  0  3
 0  0  0  0

```



提示 在上述还原操作中, 如果在结果与原矩阵是否相等的对比中出现了 `false`, 一般是因为浮点的计算精度问题, 实际上是正确还原的, 有兴趣的读者可显示具体结果分别查看。

利用好矩阵因子分解方法, 可以在求线性解或矩阵幂等问题时提升性能或降低内存消耗。除了上述介绍的这些用法, Julia 还提供了 `factorize()` 函数, 能够根据矩阵的类型, 选择更有效的分解方法, 例如:

```

julia> using LinearAlgebra

julia> A = [1.5 2 -4; 3 -1 -6; -10 2.3 4]
3×3 Array{Float64,2}:
 1.5  2.0 -4.0
 3.0 -1.0 -6.0

```

```

-10.0  2.3  4.0

julia> factorize(A)
LU{Float64,Array{Float64,2}}
L factor:
3×3 Array{Float64,2}:
 1.0  0.0  0.0
-0.15 1.0  0.0
-0.3  -0.132196 1.0
U factor:
3×3 Array{Float64,2}:
-10.0  2.3  4.0
 0.0  2.345 -3.4
 0.0  0.0  -5.24947

```

因为矩阵 A 不是 Hermitian 矩阵，也不是对称矩阵、三角矩阵 (Triangular)、三对角矩阵或双对角矩阵，所以自动选择了较好的 LU 分解。再例如：

```

julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
 1.5  2.0 -4.0
 2.0 -1.0 -3.0
-4.0 -3.0  5.0

julia> factorize(B)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
3×3 Tridiagonal{Float64,Array{Float64,1}}:
-1.64286  0.0  .
 0.0     -2.8  0.0
 .        0.0  5.0
U factor:
3×3 UnitUpperTriangular{Float64,Array{Float64,2}}:
 1.0  0.142857 -0.8
 .    1.0     -0.6
 .    .       1.0
permutation:
3-element Array{Int64,1}:
 1
 2
 3

```

其中，检测出 B 是对称矩阵，所以选择了 Bunch-Kaufman 分解。

当然，对前面介绍的特殊矩阵，Julia 可以明白地根据其类型知道该矩阵的属性，或者对本身是特殊矩阵的矩阵对象进行标记，例如：

```

julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
 1.5  2.0 -4.0
 2.0 -1.0 -3.0
-4.0 -3.0  5.0

julia> sB == B
true

```

然后：


```
julia> factorize(sB)
BunchKaufman{Float64,Array{Float64,2}}
D factor:
3×3 Tridiagonal{Float64,Array{Float64,1}}:
-1.64286  0.0  .
 0.0     -2.8  0.0
  .      0.0  5.0
U factor:
3×3 UnitUpperTriangular{Float64,Array{Float64,2}}:
1.0  0.142857  -0.8
 .   1.0      -0.6
 .   .        1.0
permutation:
3-element Array{Int64,1}:
 1
 2
 3
```

还可以进行线性求解:

```
julia> x = [1; 2; 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> sB\x
3-element Array{Float64,1}:
-1.7391304347826084
-1.1086956521739126
-1.4565217391304346
```

其中,反除号运算符用于线性求解。可见这种方式易于阅读,而且能够灵活地对所有线性方程系统进行求解。



提示 更多关于分解等各种线性代数计算,读者可以自行尝试。与特殊矩阵一样,对矩阵的分解 Julia 也提供了部分的 Lapack 优化,有兴趣的读者可参阅文档详细了解。

字 符 串

人类的语言文字是由很多的字符构成的，无论词、句还是文章，实际都是字符的序列，即字符串。所以在开发中，字符串的操作极为常见，而且用途广泛。

但很多编程语言因为历史原因，对字符串并没有很好地支持，例如，性能强大的 C/C++ 语言是在后来标准库中才专门提供了处理字符串的接口。另外一个令人苦恼的地方是，对于非英语系的国家来说，更普遍的场景是需要对 ASCII 码表中 128 个字符之外的 Unicode 字符进行处理，这是包括 Python 等流行语言都捉襟见肘的地方。

Julia 自设计之初，便从根本上提供了对多语言编码的支持。在 Julia 中操作字符和字符串会非常简洁、高效、直接，同时还兼容了类 C 风格的字符串操作方式。

9.1 字符

字符类型是前面第 6 章介绍类型系统时提到的 Char 类型，是一个 32 位存储结构的元类型，父类型为 AbstractChar 抽象类型。若要在 Julia 中表述单一字符，只需将字符放在单引号中即可，例如：

```
julia> 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> typeof(ans)
Char
```

若单引号中放多个字符，便会出错，因为语法不支持，例如：

```
julia> 'xy'
ERROR: syntax: invalid character literal
```

从上例的字符对象 'x' 创建时的提示信息便可看出，类型系统中的 Char 类型已经支

持了 Unicode 字符集 (UTF-8 编码), 所以对于非 ASCII 码的任意 UTF-8 字符, 也同样可以创建字符对象, 例如:

```
julia> '①'
'①': Unicode U+2460 (category No: Number, other)

julia> '1/4'
'1/4': Unicode U+00bc (category No: Number, other)

julia> '¥'
'¥': Unicode U+00a5 (category Sc: Symbol, currency)
```

成功后, 除了提示该字符的 Unicode 码值外, 还告知其在字符集中的分类。

当然, 我们也可以使用 Unicode 码值创建字符对象, 只需以 \u 或 \U 作为前缀输入其十六位码值即可, 例如:

```
julia> '\u0'
'\0': ASCII/Unicode U+0000 (category Cc: Other, control) # 提示中有ASCII字样, 表示属于该字符集

julia> '\u78'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> '\U2200'
'V': Unicode U+2200 (category Sm: Symbol, math)

julia> '\U10ffff'
'\U10ffff': Unicode U+10ffff (category Cn: Other, not assigned) #无效字符
```

其中, 字符 'V' 的 Unicode 码值是 2200, 所以 '\U2200' 或 '\u2200' 都可以指代这个字符。

前述的 Unicode 码值是以十六进制表示的, 不过在 Julia 中, 如果使用 Char 的构造函数创建对象, 可以使用其他常见的进制, 包括十进制、二进制与八进制。例如:

```
julia> Char(0x78) # 十六进制
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> Char(120) # 十进制
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> Char(0b01111000) # 二进制
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> Char(0o170) # 八进制
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

不过并不是所有的整型都是有效的 Unicode 码值, 所以给定的 Unicode 码值未必能够获得有效的字符。但为了性能, Char() 在转换时并不会检查有效性。如果获知有效性, 可显式调用 isvalid() 函数, 例如:

```
julia> Char(0x110000)
'\U110000': Unicode U+110000 (category Cn: Other, not assigned)
```

```
julia> isvalid(Char, 0x110000)
false
```

事实上,无论采用何种进制,都是描述的方式不同而已,在内存表达上都是一致的。所以任意字符的实际编码值,Julia 均允许转换到 Number 类型的任意一个具体子类型。例如:

```
julia> convert{Int64, 'x'}
120
```

```
julia> convert{Float64, 'x'}
120.0
```

```
julia> convert{Rational, 'x'}
0x000000078//0x000000001
```

```
julia> convert{Complex, 'x'}
0x000000078 + 0x000000000im
```

等价于:

```
julia> Int64('x')
120
```

```
julia> Float64(0x170)
120.0
```

```
julia> Complex{Im, 'x'}
0x000000078 + 0x000000000im
```

```
julia> Rational{Int64}(120)
120//1
```

```
julia> 0x000000078//0x000000001 == 120//1 # 值相同,但类型有差别
true
```

字符之间因为编码是有序的,有着前后继的关系,所以它们之间可以进行比较。例如:

```
julia> 'x' > 'A'
true
```

```
julia> 'A' <= 'a' <= 'Z'
false
```

这个并不难理解。

字符之间的四则运算除了减法,其他均是不支持的。因为减法有着明确的物理意义,表示两者编码位置的距离,但加法等其他运算是无意义的。例如:

```
julia> 'x' - 'a'
23
```

表示字符 'x' 的编码位置在 'a' 之后,它们之间相差了 23 个位置,中间有 22 个其他字符。

当然,我们可以通过在字符对象基础上加、减某个整型值,以获得该字符的后继或前继字符,例如:


```
julia> 'x' - 23
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> 'A' + 1
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
```

9.2 String 对象

字符串是连续的字符序列，在 Julia 中是 `String` 类型，而抽象类型 `AbstractString` 是其父类型，同样支持以 UTF-8 编码的 Unicode 字符集。若要自定义字符串类型，则需继承自该抽象类型。在函数调用中，如果需要字符串作为参数，最好是以 `AbstractString` 限定参数的类型，以便能够接受任意的字符串类型。



提示 同一些语言（比如 Java）相似，`AbstractString` 的字符串对象是不可变的。即一旦定义了字符串对象，其内部的元素值不能再改变，只能以其为基础生成新的字符串对象。这一点，需要开发者注意。

9.2.1 表达

代码中若以字面方式提供字符串对象，需使用成对的双引号或三组双引号表示，例如：

```
julia> "Hello World"
"Hello World"

julia> """Hello World"""
"Hello World"

julia> typeof(ans)
String
```

或者：

```
julia> s = "\u2200 x \u2203 y"
"∀ x ∃ y"
```

在使用 REPL 的过程中，如果不能正常显示字符串内容，一般需要调整控制台的配置，以便其支持 Unicode 编码。

如果字符串本来就包含双引号，则需要对内含的双引号进行转义处理，即在内部的双引号前加反斜杠 `\`，例如：

```
julia> "Contains \"quote\" characters"
"Contains \"quote\" characters"
```

或者，外部使用三组双引号，这样 Julia 就能区分内部的双引号，例如：

```
julia> """Contains "quote" characters"""
"Contains \"quote\" characters"
```

除了引号，字符串还可以容纳包括换行符等其他特殊字符，例如：

```
julia> "xxx \x1C \n\r yy"
"xxx \x1c \n\r yy"
```

需要注意的是, 换行符有三种方式[⊖]: CR、CRLF、LF, 一般分别以 \r、\r\n 及 \n 表示。但在字面字符串中使用回车输入换行时, Julia 均会以 \n 记录该换行符, 即使编辑器使用的是 CR 或 CRLF 作为换行符。如果要使用 CR 或 LF 等, 需要显式地输入 \r 或 \n 字符。

9.2.2 索引

对于字符串对象, 可以使用类似数组索引的方式取得其中任意位置的字符。例如:

```
julia> str = "Hello, world."
"Hello, world."

julia> str[1]
'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[8]
'w': ASCII/Unicode U+0077 (category Ll: Letter, lowercase)

julia> str[end]
',': ASCII/Unicode U+002e (category Po: Punctuation, other)
```

当然, 字符串与数组的索引一样, 都是从 1 开始的 (1-based)。如上所示, end 关键字在字符串中也同样适用, 表示最后一个元素的位置, 并可进行计算:

```
julia> str[end-1]
'd': ASCII/Unicode U+0064 (category Ll: Letter, lowercase)

julia> str[end+2]
',': ASCII/Unicode U+002c (category Po: Punctuation, other)
```

如果索引时越界, 则会报异常:

```
julia> str[0]
ERROR: BoundsError: attempt to access "Hello, world."
       at index [0]

julia> str[end+1]
ERROR: BoundsError: attempt to access "Hello, world."
       at index [14]
```

以数值下标的方式对字符串进行索引时, 通常能工作得很好, 但当其中存在 UTF-8 字符时会出现一些问题。例如:

```
julia> s = "\u2200x\u2203 y"
"∀x ∃ y"

julia> s[1]
'∀': Unicode U+2200 (category Sm: Symbol, math)
```

⊖ 换行 (LF) 和回车 (CR) 符来自于最早的打印机。DOS 和 Windows 一般采用 CRLF 标识新行 (换行), Unix 或 Linux 采用 LF, 而 Mac OS 则采用 CR。这些区别在文本编辑时经常遇到。

```
julia> s[2]
ERROR: UnicodeError: invalid character index 2 (0x88 is a continuation byte)

julia> s[3]
ERROR: UnicodeError: invalid character index 3 (0x80 is a continuation byte)

julia> s[4]
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> s[5]
'∫': Unicode U+2203 (category Sm: Symbol, math)

julia> s[6]
ERROR: UnicodeError: invalid character index 6 (0x88 is a continuation byte)
```

这是因为 UTF-8 是可变宽度的编码方式，并不是所有的字符都采用同样的字节数。对于码值小于 0x80（即十进制 128）的字符，仍采用 ASCII 的单字节表达；而对于码值大于等于 0x80 的字符则为多字节字符（有的甚至是 4 字节）。所以，在一个存在 UTF-8 字符的字符串中，连续字符的有效索引位置并不是连续的。如上例，∫ 是个 3 字节字符，紧随其后的字符 x 的实际位置则是 4，所以索引值 2 和 3 是无效的，这便引发了上述问题。

为了应对这个问题，Julia 提供了以下函数：

```
nextind(str::AbstractString, i::Integer, n::Integer=1) -> Int
```

当参数 $n=1$ 时，用于获得 `str` 中 i 位置之后首个 Unicode 字符的有效索引位置；当 $n>1$ 时，相当于 $n=1$ 时的多次操作；当 $n=0$ 时，会判断 i 是否为 Unicode 字符的有效起始索引，如果有效，会返回 i 值，否则报 `StringIndexError`（无效索引）或 `BoundsError`（越界）异常。

仍以 `s = "\u2200x\u2203 y"` 为例，如下：

```
julia> i1 = nextind(s, 0)           # i取0, n取1
1

julia> i2 = nextind(s, i1)          # 以前一个有效索引为参数
4

julia> i3 = nextind(s, i2)          # 以前一个有效索引为参数
5

julia> i4 = nextind(s, i3)          # 以前一个有效索引为参数
8

julia> i5 = nextind(s, i4)          # 以前一个有效索引为参数
9

julia> i5 > lastindex(s)            # 判断获得的索引是否越界
false

julia> i6 = nextind(s, i5)          # 未越界，继续取下一个有效索引
10
```

```
julia> i6 > lastindex(s)           # 已经超出s实际的字符串长度
true
```

```
julia> s[i1], s[i2], s[i3], s[i4], s[i5] # 以获得各有效索引提取字符
('V', 'x', '∃', ' ', 'y')
```

其中, 函数 `lastindex()` 用于给出字符串的最大索引值 (按字节移动), 与 `length()` 返回字符个数是不同的。

不妨再看一下 `n=0` 和 `n=2` 时的情况:

```
julia> nextind(s, 3, 0)           # 判断3是否为有效索引位置
ERROR: StringIndexError("Vx ∃ y", 3) # 无效, 报异常
```

```
julia> nextind(s, 4, 0)           # 判断4是否为有效索引位置
4                                # 有效, 返回该值
```

```
julia> nextind(s, 0, 3)           # 从0位置开始, 找到第3个有效索引位置
5
```

使用上例的方式, 我们便能够依据已知的某个有效索引值取得其后继的有效索引值, 以此获得下一个字符元素。至于 `prevind()` 函数, 因为与上例类似, 不再给出示例。

9.2.3 遍历

对于字符串的遍历, 可以使用基于索引的方式, 即:

```
julia> for i = 1:length(str)      # 或者 for i = 1:lastindex(str), 但该方式不适用于有Unicode
                                # 字符的字符串
    print(str[i], " ")
end
Hello, world.
```

但如上所述, 该方式对字符串中有 Unicode 字符的情况不适用。例如:

```
julia> s = "Vx ∃ y";
julia> for i = 1:length(s)
    println(s[i])
end
V
ERROR: UnicodeError: invalid character index 2 (0x88 is a continuation byte)

julia> for i = 1:lastindex(s)
    println(s[i])
end
V
ERROR: UnicodeError: invalid character index 2 (0x88 is a continuation byte)
```

可见, 因为 UTF-8 变字节的问题, 使得 `length()` 或 `lastindex()` 都不能获得 Unicode 字符正确的索引值。如果一定要使用索引方式, 可以使用 `eachindex()` 函数实现, 即:

```
julia> eachindex(s)
Base.StringIndex{String}("Vx ∃ y")

julia> for i in eachindex(s)
```



```

        println(i, " is ", s[i])
    end
1 is V
4 is x
5 is 3
8 is
9 is y

```

可见, 在这种方式下, `eachindex()` 函数能够正确地自动取得每个字符的索引值。

事实上, 字符串也是可迭代数集的一种, 可以基于迭代器, 采用元素迭代的方式进行遍历, 例如:

```

julia> for c in s           # 或 for c = s 或 for c in s
    println(c)
end
V
x
3
y

```

这种方式不但适用于字符串的各种情况, 无论其中是否有 Unicode 字符, 而且使用起来最为方便、简洁, 推荐选择这种遍历方式。

9.2.4 子串

若在开发中需要提取字符串中的某部分内容, 即获得其中的子串, 则可以使用范围索引的方式。例如:

```

julia> str
"Hello, world."

```

```

julia> str[2:4]
"ell"

```

```

julia> str[1:end]
"Hello, world."

```

在对字符串使用范围索引时, 有些要点需要注意:

- ❑ 范围索引使用的闭区间, 故“初始值”与“边界值”对应位置的元素都会取出。
- ❑ 如果初始值大于边界值, 结果将会是空字符串。
- ❑ 如果初始值等于边界值, 取出的将是仅有一个字符的字符串, 但不是字符。

例如:

```

julia> str[2:1]
""
# 范围无效, 取得空串

```

```

julia> str[8:8]
"w"
# 是String类型, 不是Char类型

```

9.3 变量替换

在 Julia 的字符串中，美元符号 `$` 是一个特殊的字符，用于表示紧随其后的是一个表达式或者变量名。当字符串中存在该标识符时，创建 `String` 对象前，会首先将对 `$` 标记的表达式求值，然后再用结果将原内容替换掉。这样的过程，类似于 Shell 或 Perl 脚本的变量替换机制，或被称为展开 (Interpolation)。

例如，有两个字符串对象，分别定义为：

```
julia> greet = "Hello"
"Hello"

julia> whom = "World"
"World"
```

如果有另外一个字符串，其中使用 `$` 标识符引用了这两个变量，则：

```
julia> "$greet, $whom."
"Hello, World."
```

其中，`$greet` 会被变量 `greet` 的内容（即“Hello”）代替，而 `$whom` 会被“World”替代。

再比如：

```
julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

其中，`$()` 表示其内部是一个表达式，所以在创建该字符串对象时，标识符 `$` 与表达式会一起被表达式结果所代替，即 3 替换掉了 `$(1+2)`。

还可以给出更多的例子：

```
julia> c = 'x'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

```
julia> "hi, $c"
"hi, x"
```

```
julia> v = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> "v: $v"
"v: [1, 2, 3]"
```

可见，`$` 在字符串中的展开行为，限定并不多，适用于多种常见的类型或运算。

正由于上述的 `$` 的特殊行为，如果字符串中需要使用该符号作为常规符号，则需要转义处理。例如：

```
julia> print("I have \$100 in my account.\n")
I have $100 in my account.
```

9.4 正则表达式

正则表达式在文本处理中经常会用到，同样得到了 Julia 的良好支持。在 Julia 中，正则表达式为 `Regex` 类型，表述采用了类似于字符串的方式，但其父类型并不是 `AbstractString` 类型，可视为一种非标准字符串。也正因为如此，针对字符串的一些操作对正则表达式并不适用，包括索引、遍历、字符串等。

先看一个创建 `Regex` 对象的例子：

```
julia> a = r"\s*(?:#|$) "
r"\s*(?:#|$) "
```

```
julia> typeof(a)
Regex
```

其表达方式是在常规字符串上加入前缀 `r` 作为标识，这是最为简单的方法。当然，也可以使用构造方法 `Regex()` 从一个字符串定义一个正则表达式对象，例如：

```
julia> Regex("\s*(?:#|$) ")
r"\s*(?:#|$) "
```

不过这种方式中，因为是以字符串为基础的，所以正则表达式中特有的标识符均需通过转义的方式提供。

关于正则表达式的语法规则，本书不作介绍，请参阅相关资料。下面我们介绍正则表达式的基本操作——匹配与提取。

所谓匹配，指在某个字符串中查找是否存在一个或多个子串，能够符合正则表达式对象所定义的字符构成模式（Pattern）。在 Julia 中通过调用 `occursin()` 函数便可实现，该函数的原型为：

```
occursin(r::Regex, s::AbstractString) -> Bool
```

如果 `s` 中存在子串满足给定的模式 `r`，便会返回 `true`，否则返回 `false` 值。

其用法举例如下：

```
julia> occursin(r"\s*(?:#|$)", "not a comment")
false
```

```
julia> occursin(r"\s*(?:#|$)", "# a comment")
true
```

可见这一用法还是比较简单的。

所谓提取，是将满足正则模式的多个子串内容从原字符串中单独拿出来，并能够给出每个子串的位置。通过 Julia 内置的 `match()` 函数或 `eachmatch()` 函数便可达到这些目的，而提取的信息会保存在类型名为 `RegexMatch` 的对象中。

通过 `dump()` 函数可查看此对象的内部结构，如下所示：

```
RegexMatch <: Any
 match::SubString{String}
 captures::Array{Union{SubString{String}, Nothing}, 1}
 offset::Int64
```

```
offsets::Array{Int64,1}
regex::Regex
```

其中每个成员变量的意义如下：

- ❑ 字段 `match` 记录匹配的子串整体内容。
- ❑ 字段 `offset` 记录匹配的子串整体在原字符串中的偏移。
- ❑ 字段 `captures` 记录捕捉到的各分组子串内容，即 `match` 字段中的字符串里符合圆括号内正则表达式的子串，相当于正则模式中对应的各个组（Group）；若捕捉结果为空，该字段将为 `nothing`，而 `offsets` 值会为 0（无效值）。
- ❑ 字符 `offsets` 记录捕捉到的各分组子串的偏移。
- ❑ 字段 `regex` 记录原始正则表达式的内容。

下面举例说明提取的用法，假设有字符串：

```
julia> money = "I have so much money. The cash is 2300 totally.";
```

希望提取其中的数字部分，定义正则表达式如下：

```
julia> rgx = r"cash is (\d+) totally"; # 将数字部分单独分组
```

然后调用 `match()` 函数，如下：

```
julia> result = match(rgx, money)
RegexMatch("cash is 2300 totally", 1="2300")
```

此时结果已经给出了基本的提取信息：符合模式的子串整体内容为“cash is 2300 totally”，捕捉到的数字为“2300”。

详细看一下 `result` 这个 `RegexMatch` 对象的内容：

```
julia> result.match
"cash is 2300 totally" # 匹配正则表达式的子串内容，不涉及分组问题

julia> result.offset
27 # “cash”之前有27个字符

julia> result.captures
1-element Array{Union{SubString{String}, Nothing},1}:
"2300" # 匹配子串中符合分组模式的内容（数字部分）

julia> result.offsets
1-element Array{Int64,1}:
35 # “2300”之前有35个字符

julia> result.regex
r"cash is (\d+) totally"
```

若要取得其中的数字部分，只需：

```
julia> cash = parse(Float64, result.captures[1])
2300.0
```

再看一个要求多分组的例子：

```
# 希望将字符串中的a或b、c、d三者单独提取出来
julia> m = match(r"(a|b)(c)?(d)", "hi, acd, ok")
```



```
RegexMatch("acd", 1="a", 2="c", 3="d")
```

可见，匹配的是子串“acd”，符合分组的内容也给了出来，m 的具体内容为：

```
julia> m.match
"acd"

julia> m.offset
5

julia> m.offsets
3-element Array{Int64,1}:
 5
 6
 7

julia> m.captures
3-element Array{Union{SubString{String}, Nothing},1}:
 "a"
 "c"
 "d"

julia> m.regex
r"(a|b)(c)?(d)"
```

如果要取得三个分组的信息，只需以数组索引或迭代的方式访问 `m.captures`。当然，也可以使用下述方式取得其中的元素：

```
julia> first, second, third = m.captures; first
"a"
```

需要注意的是，`match()` 函数只提取首次匹配的内容。如果原字符串中有多处可匹配，其余的都会被忽略。不过可以为该函数指定搜索的起始位置，以限定匹配的部位，例如：

```
julia> m = match(r"[0-9]", "apcqlapcq2apcq3", 1) # 从第1个字符开始
RegexMatch("1") # 找到了第一个数字

julia> m = match(r"[0-9]", "apcqlapcq2apcq3", 6) # 从第6个字符开始
RegexMatch("2") # 找到了第二个数字

julia> m = match(r"[0-9]", "apcqlapcq2apcq3", 11) # 从第11个字符开始
RegexMatch("3") # 找到了第三个数字
```

如果要将符合要求的所有内容都一次性地提取出来，可以考虑使用 `eachmatch()` 函数。该函数会将每个匹配部位处得到的 `RegexMatch` 封装在一个迭代器中。例如：

```
julia> n = eachmatch(r"\d", "apcqlapcq2apcq3")
Base.RegexMatchIterator{RegexMatch{String}}(r"\d", "apcqlapcq2apcq3", false)

julia> for i in n
    println(i)
end
RegexMatch("1")
RegexMatch("2")
RegexMatch("3")
```

再看有分组的情况，如下：


```
julia> String(['a', 'b', 'c'])           # 字符数组
"abc"
```

其中, 可直接使用 `String` 构造函数对字符数组进行连接操作, 从而创建新的字符串。

但这种方式不支持字符的元组结构:

```
julia> String( ('a', 'b', 'c') )         # 字符元组
ERROR: MethodError: no method matching String(::Tuple{Char,Char,Char})
Closest candidates are:
  String(::String) at boot.jl:309
  String(::Array{UInt8,1}) at strings/string.jl:39
  String(::Base.CodeUnits{UInt8,String}) at strings/string.jl:75
...
```

也不支持从字符串数组创建, 例如:

```
julia> String(["abc", "def"])             # 字符串数组
ERROR: MethodError: no method matching String(::Array{String,1})
Closest candidates are:
  String(::String) at boot.jl:309
  String(::Array{UInt8,1}) at strings/string.jl:39
  String(::Base.CodeUnits{UInt8,String}) at strings/string.jl:75
...
```

这是笔者认为可以改进的两个地方。

我们可以使用专门的连接函数 `join()` 来达到目的, 例如:

```
julia> join(['a','b','c'])                # 字符元组
"abc"

julia> join(["abc","def"])                # 字符串数组
"abcdef"

julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
"apples, bananas and pineapples"
```

也适用于 `Set` 对象, 例如:

```
julia> a = Set(['a','b','c'])
Set{Char}(['b', 'a', 'c'])

julia> join(a)
"bac"
```

因为 `Set` 是无序的, 所以结果字符串的内容是难以预料的。

还有一点, 在调用 `join()` 函数时, 不要直接将待连接的内容以独立字符串或字符参数的形式传入, 否则会得到不可预期的结果, 例如:

```
julia> join('a','b','c')
"a"                                     # 得到的结果并不是预期的abc

julia> join("abc","def")
"adefbdefc"                           # 得到的结果并不是预期的abcdef
```

但是, 函数 `string()` 是可以这么做的, 即:

```
julia> string('a','b','c')
```

```
"abc"
```

```
julia> string("abc","def")
"abcdef"
```

注意该函数的首字母是小写的。

但函数 `string()` 对迭代结构的处理与 `join()` 函数是不同的, 例如:

```
julia> string(['a','b','c'])
"['a', 'b', 'c']"
```

```
julia> string( ('a','b','c') )
"('a', 'b', 'c')"
```

```
julia> string( Set(['a','b','c']) )
"Set(['b', 'a', 'c'])"
```

它会直接将结构转为字符串, 而不是将内部元素进行连接。

除了使用函数外, 也可以通过星号 `*` 将其前后的字符串 (或者字符) 进行连接, 例如:

```
julia> "abc" * "def" # 字符串连接字符串
"abcdef"
```

```
julia> greet = "Hello"; whom = "world";
```

```
julia> greet * ", " * whom * ".\n" # 支持变量内容展开
"Hello, world.\n"
```

```
julia> 'a' * 'b' # 字符连接字符
"ab"
```

```
julia> 'a' * "bc" # 字符连接字符串
"abc"
```

```
julia> "ab" * 'c' # 字符串连接字符
"abc"
```

但不能认为可以通过 `map()` 函数的方式实现, 得到的结果并不是预期的:

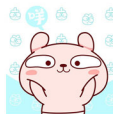
```
julia> map(*, "abc", "def")
3-element Array{String,1}:
"ad"
"be"
"cf"
```

不过可以使用 `broadcast()` 实现多个字符串的连接。例如:

```
julia> broadcast(*, "abc", "def")
"abcdef"

julia> broadcast(*, "ab", 'c', 'd', "ef")
1-element Array{String,1}:
"abcdef"
```

从一定意义上讲, 连接字符串的操作符 `*` 并不是乘法运算符, 所以不能利用 `map()` 函数的特性。至于为何不像其他语言通过加号 `+` 的方式连接字符串, Julia 语言的设计者有着



自己的道理，有兴趣的读者可以参考相关文献。

在字符串创建方法中，还有一些辅助函数可以使用，例如 `repeat()` 函数，可以构造内容重复出现的字符串：

```
julia> repeat(".:Z:.", 10)
".:Z:...Z:...Z:...Z:...Z:...Z:...Z:...Z:...Z:.."
```

更多有趣的函数，读者可以参阅官方文档。

9.5.2 比较

字符串是字符序列，所以可以像字符那样执行比较等运算（遵循字典原则），例如：

```
julia> "abracadabra" < "xylophone"
true

julia> "abracadabra" == "xylophone"
false

julia> "Hello, world." != "Goodbye, world."
true
```

但字符串与字符之间是不能进行大小比较的，例如：

```
julia> "abc" < 'a'
ERROR: MethodError: no method matching isless(::String, ::Char)

julia> 'a' > "abc"
ERROR: MethodError: no method matching isless(::String, ::Char)
```

但是，是否相等或者是否相同运算符是可以使用的，即：

```
julia> "abc" != 'a'
true

julia> "abc" == 'a'
false

julia> "abc" === 'a'
false
```

这是因为，是否相等或是否相同运算符本身就不限定操作数的类型。

9.5.3 搜索

如果要判断某个字符串是否是另一个的字符串，使用内置的 `occursin()` 函数即可。例如：

```
julia> str = "Hello,World. There are 50 billion people in the World";

julia> occursin("orld", str)
true

julia> occursin(str[2:4], str)
true

julia> occursin(str[1:end], str)
```



```
true
```

```
julia> occursin("why", str)
false
```

```
julia> occursin("H", str)
true
```

该函数同样适用于针对 Char 对象的判断，例如：

```
julia> occursin('H', str)
true
```

当然对于这种要判断字符是否属于某个字符串的情况，也可以使用 in() 函数，例如：

```
julia> in('W', str)
true
```

```
julia> in('y', str)
false
```

也可使用 in 的断言方式，即：

```
julia> 'W' in str
true
```

```
julia> 'y' in str
false
```

或者：

```
julia> 'H' ∈ str
true
```

```
julia> 'W' ∉ str
false
```

如果通过 occursin() 函数判断某个字符串属于另外一个字符串，或者使用 in 判断某个字符属于一个字符串后，仍希望获知它们处于包含字符串的位置时，可使用以下两个函数对包含字符串进行搜索：

```
findfirst(chars, str)    # 从前往后在str中搜索chars内容
findlast(chars, str)     # 从后往前在str中搜索chars内容
```

其中，chars 是待搜索内容，可以是 AbstractString 类型，也可以是 Regex 对象（正则表达式），但不支持字符 Char 类型。

函数 findfirst() 会从前往后搜索 str 的内容，当找到 chars 时便会返回 chars 在 str 中的位置，而不再对后续内容进行搜索。所以 findfirst() 函数只会给出 chars 出现的第一次位置。相对地，函数 findlast() 会从后往前搜索，并给出 chars 首次出现的问题。在使用中，这两个函数会以 Range 对象的方式给出发现内容的首尾位置。例如：

```
julia> findfirst("Hello", str)
1:5
```

```
julia> findfirst("Wor", str)
7:9
```



或

```
julia> findlast("Hello", str)
1:5
```

```
julia> findlast("Wor", str)      # 返回的位置是第二个World中的子串位置
49:51
```

也可以使用正则表达式：

```
julia> findfirst(r"\d{2}", str) # 搜索str中首次出现的两个连续数字
24:25
```

显然，我们以返回的 Range 对象对被搜索的原字符串提取子串，结果子串应该与搜索的目标串具有一致性（如果是字符串，那么应该相同），例如：

```
julia> str[1:5] == "Hello"
true
```

```
julia> str[49:51] == "Wor"
true
```

如果这两个函数搜索失败，均会返回 nothing 对象，即：

```
julia> findfirst("some", str) == nothing
true
```

```
julia> findlast("some", str) == nothing
true
```

对于 Unicode 字符串，该函数同样适用，例如：

```
julia> s = "∀x∃ y";
```

```
julia> s[5:5]
"∃"
```

```
julia> findfirst("∃ y", s)
5:9
```

```
julia> s[5:9]
"∃ y"
```

不过因为 UTF-8 是多字节字符的缘故，所以返回的位置与普通字符会有所差别。

另外还有两个函数 findnext() 及 findprev() 也具有类似的功能，区别在于能够指定搜索的起始位置，这样便能够通过起始位置不断地执行搜索任务，直至找到所有的子串或模式。关于这两个函数的具体用法这里不再继续介绍，读者可参阅官方文档。

9.5.4 替换

将原字符串中符合要求的子串替换为新的内容，这也是经常会遇到的情况。在 Julia 中，使用内置函数 replace() 便可实现，其原型为：

```
replace(str::AbstractString, pattern=>rpl; count)
```

该函数会将匹配上的内容进行替换。其中，str 为原字符串；pattern 是 Char、String



或正则表达式类型（描述欲替换内容的模式）；`rpl` 是用于替换的新内容，可以是 `String`、`Char` 或一个函数对象，当 `rpl` 为一个函数时，`pattern` 匹配的子串内容 `s` 会被 `rpl(s)` 的计算结果所代替；参数 `count` 为可选，用于指定从左到右需要替换的个数，不设置时会将所有匹配上的内容替换，否则会替换掉 `count` 个内容。

举例说明如下：

```
julia> str = "Hello,World.";

julia> s = replace(str, "o"=>"p")          # 将str中所有的字符o替换为p
"Hellp,Wprld."

julia> s = replace(str, 'o'=>'p')          # 对单字符情况也可直接使用Char类型参数
"Hellp,Wprld."

julia> str
"Hello,World."
```

注意原字符串 `str` 本身是不会变的，如前文所述，`String` 类型是不可变的，所以 `replace()` 会创建新的字符串对象。再看一下设置了 `count` 的情况：

```
julia> s = replace(str, 'o'=>'p'; count=0)
"Hello,World."          # 没有内容被替换

julia> s = replace(str, 'o'=>'p'; count=1)
"Hellp,World."          # 第一个o被替换

julia> s = replace(str, 'o'=>'p'; count=2)
"Hellp,Wprld."          # 两个o均被替换

julia> s = replace(str, 'o'=>'p'; count=3)  # count比实际个数多
"Hellp,Wprld."          # 两个o均被替换
```

该函数也支持正则表达式作为搜索参数 `pattern`，例如：

```
julia> replace("I have 279 money. The cash is 2300 totally.", r"\d+"=>"8888")
"I have 8888 money. The cash is 8888 totally."
```

其中，所有的数字子串替换为指定的数值。

当然，也可以对符合模式要求的子串实施函数变换，例如：

```
julia> s = replace(str, r"o"=>uppercase)    # 将str中符合模式 r"o"的内容均置为大写
"Hello,World."
```

其中，对所有的小写 `o` 使用 `uppercase()` 函数，将其转为大写。

9.5.5 分割

所谓分割，以某个 `Char`、`String` 或正则表达式的对象为依据，将原字符串拆分为多个字符串的过程。在以字符串形式进行数据交换的应用中，分割经常出现。

Julia 中的 `split()` 函数可用于字符串的分割，其原型为：

```
split(str::AbstractString, [chars]; limit::Integer=0, keep::Bool=true)
```



其中, `str` 是待分割字符串; `chars` 是 `Char`、`String` 或正则表达式的对象, 描述分隔的依据, 如果不指定, 则默认为空格, 但需注意 `chars` 表达的内容一定要在 `str` 中能够找到, 否则不会出现预期的结果; 另外两个参数是可选的键值参数, 分别用于限定结果的数量以及设定是否保留空的内容。

例如, 如下两个字符串:

```
julia> s1 = "hi,hello,hey,world";
julia> s2 = "hi hello hey world";
```

分别调用 `split()` 函数, 以逗号与空格对其进行分割, 结果如下:

```
julia> a = split(s1, ',') # 以逗号分割
4-element Array{SubString{String},1}:
"hi"
"hello"
"hey"
"world"
```

```
julia> b = split(s2) # 以空格分割
4-element Array{SubString{String},1}:
"hi"
"hello"
"hey"
"world"
```

确实得到了被分离开的字符串数组。

值得注意的是, 分离出的字符串类型不是 `String`, 而是 `SubString{String}`, 结构为:

```
SubString{String} <: AbstractString
string::String          # 分割前的原字符串
offset::Int64           # 当前部分的位置相对于在原字符串中首字符的偏移量
ncodeunits::Int64      # 子串的字符数
```

这其实是相对于 `String` 类型的视图。上例中的 `a[1] = "hi"` 是 `SubString{String}` 类型, 其结构中的内容为:

```
julia> dump(a[1])
SubString{String}
  string: String "hi,hello,hey,world"
  offset: Int64 0
  ncodeunits: Int64 2
```

可以使用构造方法 `String()` 将该类型转为 `String` 类型, 即:

```
julia> String(a[1])
"hi"
```

```
julia> typeof(ans)
String
```

再看两个例子, 如下:

```
julia> s3 = "hi 1 hello 2 hey 3 world";
julia> s4 = "hi ① hello ① hey ① world No";
```

分别以正则表达式与 Unicode 字符分割, 结果为:



```
julia> split(s3, r"\d")           # 以单个数字为分割符
4-element Array{SubString{String},1}:
"hi "
" hello "
" hey "
" world"

julia> split(s4, "①")           # Unicode字符作为分隔符
4-element Array{SubString{String},1}:
"hi "
" hello "
" hey "
" world No"
```

可见, Julia 的 `split()` 适用场景很广泛, 能够灵活地以任何有效字符对字符串进行分割。



提示 对于字符串的操作, 还有不少本书不能尽述的, 可参见附录 C。

9.6 字节数组

如果在双引号或三引号表示的字面值之前加上字符 `b` 作为前缀标识符, 之后的字符串内容便会被 Julia 解析为字节数组 (Byte Array Literals), 即 `Array{UInt8,1}` 类型的对象。例如:

```
julia> b"bytes"
5-element Array{UInt8,1}:
0x62
0x79
0x74
0x65
0x73
```

```
julia> b"""bytes"""
5-element Array{UInt8,1}:
0x62
0x79
0x74
0x65
0x73
```

若需将某个字符串对象转为字节数组, 可显式地进行类型转换, 例如:

```
julia> s = "bytes";

julia> Array{UInt8, 1}(s)
5-element Array{UInt8,1}:
0x62
0x79
0x74
0x65
0x73
```



或者使用 `convert()` 函数，即：

```
julia> convert(Array{UInt8, 1}, "bytes")
5-element Array{UInt8,1}:
 0x62
 0x79
 0x74
 0x65
 0x73
```

在将字符串转为字节数组的过程中，Julia 遵循以下的转换规则：

- ❑ ASCII 字符或转义字符作为单字节处理。
- ❑ 数制前缀（如 `\x` 或 `\o`）生成对应值的字节序列。
- ❑ Unicode 前缀会输出 UTF-8 码值的字节序列。

举例说明：

```
julia> b"DAT\xff\u2200"
7-element Array{UInt8,1}:
 0x44      # 字符D的ASCII码值，即十进制值68
 0x41      # 字符A的ASCII码值，即十进制值65
 0x54      # 字符T的ASCII码值，即十进制值84
 0xff
 0xe2      # Unicode字符\u2200的UTF-8编码的三字节之一，十进制值为226
 0x88      # UTF-8码值之一，即十进制值136
 0x80      # UTF-8码值之一，即十进制值128
```

其中，字符串“DAT”被转为其 ASCII 码值；十六进制值 `\xff` 保持不变并对应一个字节，因为其在 `UInt8` 表达范围内；但 Unicode 字符 `\u2200` 会按其 UTF-8 编码转成三个字节。

需注意 `\xff` 与 `\uff` 这两类表述的差别：前者为单字节值 255，后者表达的是 UTF-8 码值的 255。所以在将它们转为字节数组时结果不会相同，后者 `\uff` 会被转为两个字节，即：

```
julia> b"\xff"
1-element Array{UInt8,1}:
 0xff

julia> b"\uff"
2-element Array{UInt8,1}:
 0xc3
 0xbf
```

另外，`\xff` 这类表述在字符与字符串中的意义是不同的：在字符中本质意义是码值，但在字符串中会被直接视为普通的数值（即字节值）。若要在字符串中表达编码值，最好使用 `\u` 或 `\U` 前缀，这些码值会在转换中变成多字节序列。

对于码值小于 `\u80` 的字符，因为其 UTF-8 码值正好与 `\x` 表示的数值一致，所以它们都会被转为相同的单字节值，例如：

```
julia> b"\x20"
1-element Array{UInt8,1}:
```



```
0x20
julia> b"\u20"
1-element Array{UInt8,1}:
 0x20
```

所以此时 `\x` 与 `\u` 两个表述的差别是可以忽略的。

关于字节数组的具体应用，将在第 12 章介绍，此处不再赘述。

9.7 与数值的转换

在开发中，字符串与数值之间的相互转换是经常遇到的，为此单独用此节内容讨论这个问题。在 Julia 中，将数值转为字符串非常方便，只要调用 `string()` 函数即可。举例说明如下：

```
julia> string(10)
"10"

julia> string(0b100101)
"37"

julia> string(1//3)
"1//3"

julia> string(2.0+3.0im)
"2.0 + 3.0im"

julia> string(pi)
"π = 3.1415926535897..."
```

而且，当输入数值是整型时，还可以通过 `base` 参数说明数值的数制，例如：

```
julia> string(328, base=8)
"510"

julia> string(100101, base=2)
"11000011100000101"

julia> string(0x9ac8b, base=16)
"9ac8b"
```

或者，在需要时，通过设定 `pad` 参数对输出的字符串进行对齐：

```
julia> string(328, base=8, pad=6)
"000510"

julia> string(100101, base=2, pad=20)
"00011000011100000101"

julia> string(0x9ac8b, base=16, pad=10)
"000009ac8b"
```

当转换的字符串中字符数不足 `pad` 指定的数量时，会在头部补充 0。



反之，若需将字符串转为数值，可通过 `parse(type, str; base=10)` 函数，其中，`type` 用于指定结果的数据类型，`str` 是待解析的字符串，`base` 用于说明数制，默认为 10 进制。例如：

```
julia> parse{Int64, "3043", base=5}    # 将5进制数值3043转为Int64整型的十进制数值398
398

julia> parse{Int32, "248", base=13}    # 将13进制数值248转为Int32整型的十进制数值398
398

julia> parse{Float32, "38.2"}
38.2f0

julia> parse{Complex{Float64}, "3.2e-1 + 4.5im"}
0.32 + 4.5im
```

有了 `string()` 和 `parse()` 在两者转换间的上述方法，我们便不再需要像使用其他语言那样，为了这些小小的问题，大费周章地写一堆辅助函数了，能够将精力放在业务逻辑的实现上。



元编程

Julia 语言作为后起之秀，博采众长，吸取了其他各种优秀语言的优势，符合现代编程语言的发展潮流。从前面的学习中，我们能够深切地感受到，Julia 语言不仅足够灵活强大，而且开发效率很高。本章要介绍的是 Julia 语言的另外一个强大机制——元编程^①。

类似于 Lisp 语言，Julia 源代码本身也是其类型系统的一部分，同样有着自己的数据类型，与语言中常规的类型有着同样的地位。故此，Julia 语言便有了神奇的能力——可以像操纵整型、字符串、数组等常规对象一样，创建并控制代码片段。

Julia 以非常自然的方式支持了元编程范式，借助强大的反射能力，能够不需要额外的构建步骤，便可生成精妙高效的代码片段（称为元程序）；而且可以得到与 Lisp 相近的真实宏系统，即在抽象语法树^②层面实现各种操作，而不是像 C/C++ 语言那样通过预编译进行文本内容替换的“宏”定义。

下面我们从基本的符号类型开始，逐步介绍 Julia 元编程的机制。

10.1 Symbol 类型

Julia 的源代码由各种可执行的表达式构成，而表达式语句又由多种语言要素构成，包

① 元编程（Meta-programming）是指，某个语言可以将其他语言作为内部可操作的对象，从而能在运行期动态生成代码片段（也称为元程序），巧妙地实现一些功能。其中，编写元程序的语言被称为元语言，被操作语言为目标语言。以自身为目标语言的能力被称为反射。

② 抽象语法树（Abstract Syntax Tree, AST）是对源代码语法结构抽象所得到的一种树状结构。作为程序编译的中间形式，可方便地用于开发各种源程序处理工具，在程序分析等领域有广泛的应用，例如源程序浏览器、智能编辑器、语言翻译器等。

括立即数、变量或对象名、运算符、界定操作符（比如圆括号、索引中括号等）以及元素分割符等等。这些基本的语言要素，事实上都是某种符号，是 Julia 中 `Symbol` 类型的实例对象。

在表达中，`Symbol` 类型（即符号类型）是一组以冒号 `:` 作为前缀标识符的符号组合，可以描述源代码的结构单元，是一种不可变的驻留字符串[⊖]。符号类型是 `DataType` 的实例之一，所以也是 Julia 类型系统的一部分，故也可像普通类型那样进行各种操作，甚至任意地组合。

例如，表达式 `a = (1+2)*3` 便可拆解为如下的几个基本的符号对象：

```
:a  :(:)  :()  :1  :+  :2  :*  :3
```

其中，赋值号 `=` 因为存在歧义，所以加上了圆括号以做限定。

可以通过 `Symbol` 的构造方法将它们重新组装，例如：

```
julia> Symbol(:a, :(:), :1, :+, :2, :*, :3)
Symbol("a=1+2*3")
```

```
julia> Symbol(:a, :(:), :2, :*, :3, :+, :1)
Symbol("a=2*3+1")
```

不仅如此，还可以将符号对象与字符串等其他类型混合在一起，例如：

```
julia> Symbol("(", :a, :+, :1, ")")
Symbol("(a+1)*(2+3)")
```

```
julia> Symbol("(", :a, :var, :+, :10, ")")
Symbol("(avar+10)/(2+15)")
```

不仅是常规类型，复合类型也可以作为符号对象，例如：

```
julia> :[10,20,30]
:([10, 20, 30])
```

```
julia> :(10,20,30)
:((10, 20, 30))
```

```
julia> :(Set([1,2,3,4]))
:(Set([1, 2, 3, 4]))
```

其中，集合对象 `Set([1,2,3,4])` 由于涉及运算符或界定符，需加上界定符圆括号将对象组合在一起。

既然符号类型是 Julia 的内部可操作类型，自然可以将其绑定到某个变量上，例如：

```
julia> a = :foo
:foo
```

```
julia> b = :[]
:([])
```

⊖ 字符串驻留 (Interned String) 是一种内存管理方法。对于字面值字符串，会先到常量池中获取，有则返回无则创建，以避免频繁的创建和销毁操作。

10.2 Expr 类型

虽然 Symbol 可以描述表达式的结构，但仍是不可执行的，而下面介绍的 Expr 类型是能够对构造的对象进行求值 (Evaluation) 操作的。

10.2.1 构造

冒号标识符除了可定义 Symbol 类型外，还可以定义可执行的源代码结构对象，即 Expr 类型。例如：

```
julia> b = :(1+2)
:(1 + 2)

julia> typeof(b)
Expr
```

其中，1+2 虽然被冒号标识，但创建的对象并非 Symbol 类型，而是被绑定到变量 b 的 Expr 类型。

Julia 能够自动识别某个被标识的表达式或其结构单元是否可执行，并创建对应的类型。我们可以使用函数 dump() 查看 Expr 对象 b 的内部结构，如下：

```
julia> dump(b)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 2

julia> b.head
:call

julia> b.args
3-element Array{Any,1}:
  :+
  1
  2
```

可见，Expr 对象主要有两个成员字段，head 与 args。其中 head 是一个 Symbol 类型，用于标识对象的类别，例如 b 是 call 类；数组 args 则按序记录表达式的所有元素，包括运算符、操作数等，例如 b 的元素有作为 Symbol 对象的 + 运算符，还有参与加法的两个 Int64 类型的数值 1 和 2。

字段 head 若是 call 类别的 Expr 对象便是可执行的代码单元，只需调用 Julia 的内置函数 eval() 即可对其计算，求得其结果：

```
julia> eval(b)
3
```

可见，计算结果正是预期的 3，而且该调用过程十分简单。

对于上节中的表达式 $a = (1+2)*3$ ，同样可以创建为 Expr 对象，即：

```
julia> x = :(a = (1+2)*3)
:(a = (1 + 2) * 3)
```

之后，便可调用 eval() 函数求值：

```
julia> eval(x)
9
```

```
julia> a
9
```

事实上，Expr 对象中的 head 是区别于不可求值的 Symbol 对象的重大差别之一。该字段一个很重要的作用便是，告知 eval() 函数应该进行怎样的计算。所以，我们可以通过 Expr 的构造方法，将不可计算的 Symbol 对象作为参数，并将运算符传入填充其 head 字段，便可创建出可执行的 Expr 对象。例如：

```
julia> y1 = Expr(:call, :+, :1, :2)
:(1 + 2)
```

其中，Expr 的参数都是 Symbol 类型，但 + 运算符是函数的一种，所以其首个参数为 call 类别。

对上例中的 y1 调用 eval()，便能顺利求得其结果，如下：

```
julia> eval(y1)
3
```

基于此，我们可以用 Expr 将 $a = (1+2)*3$ 的基本 Symbol 对象重新封装，如下：

```
julia> y2 = Expr(:call, :*, y1, :3)
:((1 + 2) * 3)
```

```
julia> y3 = Expr(:(=), :a, y2)
:(a = (1 + 2) * 3)
```

```
julia> typeof(y3)
Expr
```

其中， $(1+2)$ 部分已经绑定到变量 y1 中，不再重复。至此，我们得到了绑定到 y3 的 Expr 对象，而且其 head 是 $:(=)$ 而不是 $:call$ ，因为这不是一个函数调用，而是一个赋值操作。

先对比一下 y3 与 $x = :(a = (1+2)*3)$ 的差别，即：

```
julia> y3 == x
true
```

可见，是内容相等的两个 Expr 对象。

对 y3 调用 eval() 函数，结果为：

```
julia> eval(y3)
9
```

```
julia> a
```

9

可见得到了预期的结果，`a` 被成功赋值为 $(1+2)*3$ 的计算结果。

下面我们回顾一下之前的过程。

首先，`Expr` 对象可以通过可变参数的构造方法创建，且参数可以是 `Symbol` 对象，也可以是另一个 `Expr` 对象（即嵌套）；若是表达式或变量，则会先计算或替换，再创建 `Expr` 对象。例如：

```
julia> x = 1;

julia> c = Expr(:call, :+, x, (1+1))
:(1 + 2)
```

而且变量可以使用类型字符串中的方式，以美元符 `$` 标识，例如：

```
julia> c = :($x+$ (1+1))
:(1 + 2)
```

其次，`Expr` 对象有三个基本要素：一是运算符、函数或其他可执行的操作符（例如赋值符号），且需以 `Symbol` 对象的方式在构造方法的第二个参数中提供；二是以 `Symbol` 对象的方式在第一个参数中提供 `Expr` 对象的类别，即可执行操作的类别，是 `:call` 这种函数调用，还是 `: (=)` 这种赋值操作；三是在后续参数中提供参与计算执行的操作数。

例如 `Expr(:call, :+, :1, :2)` 中，加号是运算符也是函数的一种，所以类别是函数调用，而后续的 `:1` 与 `:2` 是参与加法的操作数。

最后，`Expr` 对象是有嵌套关系的，每一层都需包括操作类别、操作符或函数以及操作数。可以认为，一个操作便需要一层 `Expr` 结构。为了说明这个问题，我们可以使用 `dump()` 函数查看 `:(a = (1+2)*3)` 的内部结构，如下所示：

```
julia> dump(:(a = (1+2)*3))
Expr                               # 第1层
  head: Symbol =                   # 第1层操作类别: =
  args: Array{Any}((2,))
    1: Symbol a                    # 第1层操作方式: 赋值对象a
    2: Expr                        # 第2层, 即第1层的右操作数
      head: Symbol call           # 第2层操作类别: 函数调用
      args: Array{Any}((3,))
        1: Symbol *               # 第2层操作方式: *
        2: Expr                   # 第3层, 即第2层操作数之一
          head: Symbol call       # 第3层操作类别: 函数调用
          args: Array{Any}((3,))
            1: Symbol +           # 第3层操作方式: +
            2: Int64 1            # 第3层操作数1
            3: Int64 2            # 第3层操作数2
          3: Int64 3              # 第2层操作数3
```

这种嵌套结构经常会出现表达式中有多重计算的情况中，我们一般不需要做过多深究，因为 Julia 会在内部自动创建这样的嵌套结构。如果要更清晰地查看一个 `Expr` 对象的层次结构，可以使用 `Meta.show_sexpr()` 函数，例如：

```
julia> Meta.show_sexpr(:(a = (1+2)*3))
```

```
(: (=), :a, (:call, :*, (:call, :+, 1, 2), 3))
```

该函数会将用圆括号的层次方式简洁地将 Expr 对象的内部结构表现出来。

Expr 类型可视为一种特别的字符结构，有点类似字符串，但不是字符串。不过 Expr 对象可以通过 string() 函数（注意大小写）很容易转换为 String 对象。例如：

```
julia> string(:(a = (1+2)*3))
"a = (1 + 2) * 3"
```

但不能通过 String 的构造方法实现该功能。

同样，字符串也不能通过 Expr 的构造方法创建为 Expr 对象，而要使用 Meta.parse() 函数，例如：

```
julia> Meta.parse("1+2")
:(1 + 2)

julia> Meta.parse("a=(1+2)*3")
:(a = (1 + 2) * 3)

julia> typeof(ans)
Expr
```

对于复杂的表达式，例如多个语句构成的复合表达式，如下：

```
julia> x = 1;

julia> y = 2;

julia> swap = begin                # 交换x与y中数值的复合表达式
                z = x
                x = y
                y = z
            end;

julia> x
2

julia> y
1
```

其中，begin 结构定义了一个复合表达式对象，有变量 swap 指代，用于交换变量 x 与 y 的内容。

若要将这种复合语句转为 Expr 对象，可以通过 Julia 提供的“引述”结构实现。所谓引述，即使用 quote 与 end 两个关键字，将多个语句封装成 Expr 对象，即：

```
julia> ex = quote
                z = x
                x = y
                y = z
            end;

julia> typeof(ex)
Expr
```

```
julia> ex
quote
# REPL[33], line 2:
z = x
# REPL[33], line 3:
x = y
# REPL[33], line 4:
y = z
end
```

通过这种方式创建的 Expr 对象与上述的略有不同，除了有对应的表达式语句外，还带有语句所在模块与在源代码中的行位置等信息。调用 dump() 查看其结构，结果为：

```
julia> dump(ex)
Expr
head: Symbol block # 第1层，代码块类型
args: Array{Any}((6,))
 1:LineNumberNode
   line: Int64 2
   file: Symbol REPL[33]
 2:Expr
   head: Symbol = # 第2层嵌套的第1个表达式，类型为赋值
   args: Array{Any}((2,))
     1: Symbol z
     2: Symbol x
 3:LineNumberNode
   line: Int64 3
   file: Symbol REPL[33]
 4:Expr
   head: Symbol = # 第2层嵌套的第2个表达式，类型为赋值
   args: Array{Any}((2,))
     1: Symbol x
     2: Symbol y
 5:LineNumberNode
   line: Int64 4
   file: Symbol REPL[33]
 6:Expr
   head: Symbol = # 第2层嵌套的第3个表达式，类型为赋值
   args: Array{Any}((2,))
     1: Symbol y
     2: Symbol z
```

可见，这种复合表达式的 Expr 对象也是嵌套的。使用 Meta.show_sexpr() 函数会直观些：

```
julia> Meta.show_sexpr(ex)
(:block,
 (:#= REPL[33]:2 =#,
 (:=(, :z, :x),
 (:#= REPL[33]:3 =#,
 (:=(, :x, :y),
 (:#= REPL[33]:4 =#,
 (:=(, :y, :z)
)
```




虽然看起来 `quote` 与 `begin` 在语法结构上有些相似，但是有着本质的区别：`quote` 在定义时内部语句不会自动执行，仅创建了一个 `Expr` 对象，而 `begin` 在定义时，其中语句会立即执行。

10.2.2 衍生

`Expr` 对象除了能够单独运行，还可以作为函数的参数，甚至是返回值，例如：

```
julia> function derive(op, op1, op2)
    expr = Expr(:call, op, op1, op2)
end
derive (generic function with 2 methods)

julia> ex1 = derive(:+, :(1), Expr(:call, :*, 4, 5))
:(1 + 4 * 5)

julia> eval(ex1)
21
```

或者：

```
julia> ex2 = derive(:-, Expr(:call, :/, :x, :y), Expr(:call, :*, 4, 5))
:(x / y - 4 * 5)

julia> x = 12.3;

julia> y = 5.1;

julia> eval(ex2)
-17.588235294117645
```

对于相同的数值对象，只要给 `derive()` 函数传入不同的表达式参数，可以得到不同的处理过程。再比如：

```
julia> ex3 = derive(:^, :(x*y), :(z%5))
:((x * y) ^ (z % 5))

julia> z = 9;

julia> eval(ex3)
1.5484641325798407e7
```

通过改变函数 `derive()` 的定义或传入不同的参数，便可衍生出任意的表达式，从而能够实现灵活的处理，适应不同的应用场景。

在编程时，经常会遇到代码复用的情况。一些代码段会高频、反复被使用在代码文本中，为了避免冗余及进行更好的维护，我们可以使用函数、宏等方式将这类代码段进行封装，然后在调用的地方，进入引用。当然函数和宏的区别是：函数在运行期起作用，是基于栈的操作会涉及性能问题；而宏则是在编译解析时对代码进行替换，更为高效。

Julia 中，使用表达式展开然后执行 `eval()` 调用，便可达到这种代码衍生的功能，实现复用的目的。例如，对一系列操作符进行重组的代码如下：

```
# 在执行该函数前,新版会提示需要执行 import Base: +, *, &, |
for op = (:+, :*, :&, :|, :$)
    eval(quote
        ($op)(a,b,c) = ($op)((($op)(a,b),c)
    end)
end
```

进一步简化:

```
for op = (:+, :*, :&, :|, :$)
    eval(:(($op)(a,b,c) = ($op)((($op)(a,b),c)))
end
```

虽然行数少了,但一层层的括号实在难以理解,不便于维护。Julia 提供了等效于 eval() 函数的宏 @eval, 替换使用后,可以让上述代码变得更简洁:

```
for op = (:+, :*, :&, :|, :$)
    @eval ($op)(a,b,c) = ($op)((($op)(a,b),c)
end
```

宏 @eval 内部准确地包装了 eval() 的调用方式,使得代码编写更为直观。当然,对于更长的多行代码块,我们也可以使用 @eval, 如下:

```
@eval begin
    # 多行代码块
end
```

以 Expr 类型将反复使用的代码片段或者需要灵活组装的执行过程进行封装,能够让程序具备神奇的代码衍生能力,可以对一些计算过程复制转移、赋值比较或递归生成,实现各种巧妙的元程序设计。

10.3 宏

如果上文的 Expr 对象中存在变量名作为右操作数,则在求值前需先确保涉及的变量已经被有效地赋值或定义,否则会出错。而且,这样的表达式是无法对传入的变量进行限定审核的,所以无法控制结果的有效性。例如:

```
julia> e = :(a+1)
:(a + 1)

julia> a = 'A'
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

julia> eval(e)
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)

julia> a = 3
3

julia> eval(e)
4
```

其中, a 的类型可以任意,若执行 +1 操作失败,只能报错;即便是执行成功,但若类型不

是预期的，则会出现奇怪的结果。例如，`a+1` 本是为了做浮点数加法，但取被传入的字符对象，导致结果是 `Char` 类型，而不是预期的 `AbstractFloat` 类型。

Julia 中的宏 (Macro) 有着类似于函数的代码结构，可接受参数传入，不过返回的却是 `Expr` 对象。换句话说，宏建立了参数组合到某个 `Expr` 对象的映射。也正因为其有着传参的结构，所以能够更为准确地对参数进行控制，避免上述的问题。

10.3.1 定义

在 Julia 中，关键字 `macro` 用来声明一个宏类型，基本语法为：

```
macro 宏名称(参数表)
    # 具体实现
end
```

例如：

```
julia> macro sayhello()
    return :( println("Hello, world!") )    # return关键字不是必需的
end
@sayhello (macro with 1 method)
```

宏定义成功后，宏对象的名称前便会出现 `@` 前缀，以作为特有的标识。

对于已经定义的宏，可以通过 `@macroexpand1` 宏查看其真实的内部结构，例如：

```
julia> @macroexpand1 @sayhello
:((Main.println)("Hello, world!"))
```

可见，宏的结果实质上是一个 `Expr` 对象。

另外，类似于函数，宏可以定义同名的不同实现方法。例如，`@sayhello` 这个宏，之前定义了一个无参的方法，也可以再定义一个有参的方法，如下：

```
julia> macro sayhello(name)
    :(println("Hello, ", $name))    # 注意name前要使用$标识为变量
end
@sayhello (macro with 2 methods)
```

注意声明成功后的提示信息——`with 2 methods`，说明宏 `@sayhello` 有了两个方法。

需注意的是，无法使用 `methods()` 函数查看某个宏具体有多少个实现方法。这与宏本身的调用机制有关，下节讨论。

10.3.2 调用

在 Julia 中，宏获得的 `Expr` 对象会在声明时编译完成，不像 `Expr` 那样在运行期需要调用 `eval()` 函数才会执行。

宏的调用方式中有一种类似于函数，即：

```
@name(参数1, 参数2, ...)
```

其中，`name` 为宏的名称，注意 `@` 与 `name` 之间不能有空格。

宏调用还有另外一种特有的方式，即不使用圆括号界定参数，而是通过空格分开，即：

```
@name 参数1 参数2 ...
```

以上两种方法可以任选一种，没有本质上的差异。

例如，上文中的 @sayhello 宏，即：

```
julia> @sayhello()
Hello, world!
```

其内部的 Expr 对象会隐式地立即执行，计算结果将作为宏运行的结果。

该宏没有参数，若使用第二种调用方式会更为简洁，即：

```
julia> @sayhello
Hello, world!
```

再看上文定义的那个带参数的例子，即 sayhello(name) 原型。如果在调用该宏时传入了一个参数，则该方法会匹配上，而其中的 name 变量会替换为实际的内容。例如：

```
julia> @sayhello("human")
Hello, human
```

或者：

```
julia> @sayhello "human"
Hello, human
```

再看另外一个多参数的例子：

```
julia> macro twonames(name1,name2)
    :(println($name1," ", $name2))
end
@twonames (macro with 1 method)
```

然后调用，如下：

```
julia> @twonames("joy","tom")
joy tom
```

```
julia> @twonames "joy" "tom"
joy tom
```

宏的两种调用方式是不能混用的，否则会出现错误，例如：

```
julia> @twonames "joy","tom"           # 参数之间不能用逗号，只能用空格
ERROR: LoadError: MethodError: no method matching @twonames(::LineNumberNode,
::Module, ::Expr)
julia> @twonames ("joy","tom")          # 参数表与宏名之间不能有空格
ERROR: LoadError: MethodError: no method matching @twonames(::LineNumberNode,
::Module, ::Expr)
```

请注意这两种方式的差别和细节。

10.3.3 预定义宏

Julia 内部提供了很多常用的宏，下面简单介绍一下。

1. @show

该宏与 print() 或 println() 函数有着类似的功能，但是打印的方式有些不同。它

会把变量或表达式本身也展示出来。例如有如下三个变量：

```
julia> x = (1,2,3)
(1, 2, 3)
```

```
julia> y = [1 2 3]
1×3 Array{Int64,2}:
 1  2  3
```

```
julia> z = :(1+2)
:(1 + 2)
```

如果使用 `println()` 函数，效果为：

```
julia> println(x)
(1, 2, 3)
```

```
julia> println(y)
[1 2 3]
```

```
julia> println(z)
1 + 2
```

而宏 `@show` 的打印效果为：

```
julia> @show x
x = (1, 2, 3)
(1, 2, 3)
```

```
julia> @show y
y = [1 2 3]
1×3 Array{Int64,2}:
 1  2  3
```

```
julia> @show z
z = :(1 + 2)
:(1 + 2)
```

```
julia> @show x y z
x = (1, 2, 3)
y = [1 2 3]
z = :(1 + 2)
:(1 + 2)
```

2. @printf 与 @sprintf

模块 `Printf` 中的宏 `@printf` 可以将任意个数的参数，按照规定的格式打印，原型为：

```
@printf([io::IOStream],fmt, args...)
```

其中，`fmt` 为描述格式化的字符串，采用的是 C 语言的规则，具体参见相关资料；`args` 为可变参数，是待格式化的输入参数；参数 `io` 为可选参数，用于输出重定向；类型 `IOStream` 会在后文介绍。

使用示例如下：

```
julia> using Printf
```

```
julia> @printf "%x %5f money %G" 10 3.2 9.76e6
a 3.200000 money 9.76E+06

julia> @printf "%.0f %.1f %f\n" 0.5 0.025 -0.0078125
1 0.0 -0.007813 # 会四舍五入

julia> @printf "%a %f %g" Inf NaN -Inf
Inf NaN -Inf
```

其中, Inf 与 NaN 在 %a、%A、%e、%E、%f、%F、%g 或 %G 中保持不变。

我们可使用另外一个宏 @sprintf 按要求将参数格式化为字符串对象。例如:

```
julia> s = @sprintf "this is a %s %15.1f" "test" 34.567; # 尾部加分号, 让语句不自动打印结果

julia> s
"this is a test          34.6"
```

3. @assert

断言是开发调试中常用的方法, 当输入条件不成立时会上报错误, 否则不会输出任何结果。例如:

```
julia> @assert 1 == 1.0

julia> @assert 1 != 1.0
ERROR: AssertionError: 1 != 1.0

julia> @assert(1==1)

julia> @assert(1!=1)
ERROR: AssertionError: 1 != 1
```

需要注意的是, 该宏无法保证能够在编译器任意的优化级别 (Optimization Levels) 中正常地执行。而且, 官方特别地提醒, 不要将其用于密码的验证。

4. @time

该宏是一种调试工具, 可以报告表达式的运行状况, 包括耗时、内存分配等情况。例如:

```
julia> @time 1+2
0.000005 seconds (4 allocations: 160 bytes)
3 # 计算结果
```

还有一个变体 @timev, 能输出内存分配方面更详细的信息, 例如:

```
julia> @timev 1+2
0.000004 seconds (4 allocations: 160 bytes)
elapsed time (ns): 4376
bytes allocated: 160
pool allocs: 4
3 # 计算结果
```

而另外一个 @timed 宏, 可以返回更为详细的运行报告, 包括返回值、运行耗时、分配字节数、垃圾回收耗时, 以及对象的各种内存分配计数器等, 例如:

```
julia> val, t, bytes, gctime, memallocs = @timed rand(10^6);
julia> t
0.006634834
julia> bytes
8000256
julia> gctime
0.0055765
julia> fieldnames(typeof(memallocs))
(:allocd, :malloc, :realloc, :poolalloc, :bigalloc, :freecall, :total_time,
 :pause, :full_sweep)
julia> memallocs.total_time
5576500
```

有兴趣的读者可以详细了解。

如果仅仅需要耗时信息，可使用 `@elapsed` 宏，例如：

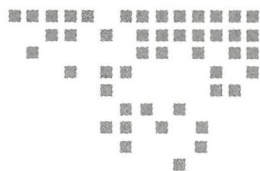
```
julia> @elapsed 1+2
2.553e-6
julia> @elapsed sleep(0.3)
0.306875862
```

该宏会报告运行耗费的秒数，而且会屏蔽运行结果。

宏 `@allocated` 仅仅输出实际内存分配的字节数，例如：

```
julia> @allocated rand(3,4).^3
384
julia> @allocated Array{Int8}(undef, 3, 4)
96
```

可见，该宏也屏蔽了运行的结果。



时间与日期

在数值计算中，数据往往具有时间维度上的特性，所以常常会涉及日期或时间方面的操作。不仅如此，各种应用场景都会涉及时间与日期的处理。为此，本章专门介绍 Julia 在日期和时间处理上的支持。

11.1 类型

时间框架涉及从年到纳秒等各种周期，而且经常会与周期转换、相距时长、早晚对比等计算相关。这些问题说复杂也不复杂，说简单也不简单，为了能够提供适用于各种场景需求的时间框架，在 Julia 的 Dates 模块（关于模块的概念参见第 13 章）中，定义了各种基础的时间类型，覆盖了常见的各种周期，如图 11-1 所示。

在图 11-1 所示的拓扑树中，左侧是右侧的父类型。可见，其中的周期类型 Period 主要分为两类，即表达日期的 DatePeriod 类型及表达时间的 TimePeriod 类型；而时间类型 TimeType 有 Date、Time 及 DateTime 三种。这些类型均是复合类型，其内部结构可以利用 dump() 函数进行查看，例如：

```
julia> import Dates
```

```
julia> dump(Week)
```

```
Week <: DatePeriod  
value::Int64
```

```
julia> dump(Microsecond)
```

```
Microsecond <: TimePeriod  
value::Int64
```

可见，这两类有着相似的结构，并均有一个类型为 Int64 的内部 value 成员变量。

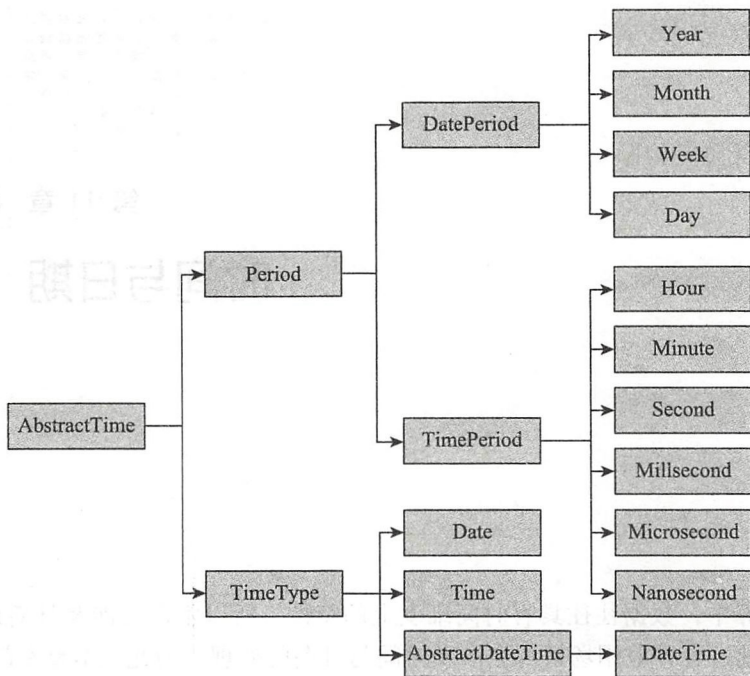


图 11-1 时间类型拓扑树

为了能够处理不同粒度的日期与详细的时间，Julia 还提供了只表达日期的 `Date` 类型、只表达时间的 `Time` 类型，以及同时表达日期和时间的 `DateTime` 类型，此三者均是抽象类型 `TimeType` 的子类型。

有关日期和各种类型的声明以及操作函数均定义在 `Dates` 中，但默认仅导出了 `Date`、`DateTime` 以及 `now()` 等基本类型与函数。所以在使用其他的类型与函数时，需要显式地在名称前加上 `Dates.` 前缀以进行引用，例如，调用 `Dates.dayofweek(dt)` 以查询日期 `dt` 是周几。当然，也可以使用 `using` 或 `import` 将其他定义引入当前空间中以省略该前缀（具体方法参见 13.1 节）。

我们可以通过 `dump()` 函数查看 `TimeType` 这三个子类型的内部结构：

```
julia> dump(Date)
Date <: TimeType
    instant:: UTInstant{Day}                                # 仅日期以Day类型记录

julia> dump(DateTime)
DateTime <: Dates.AbstractDateTime
    instant::Dates.UTInstant{Millisecond}                   # 既有日期又有时间，以毫秒记录

julia> dump(Time)
Time <: TimeType
    instant:: Nanosecond                                     # 只有时间，以纳秒记录

julia> dump(Dates.UTInstant)
```



```
UnionAll
var: TypeVar
  name: Symbol P
  lb: Core.TypeofBottom Union{}
  ub: Period <: AbstractTime
  body: Dates.UTInstant{P<: Period} <: Dates.Instant
  periods::P
```

可见它们的内部成员只有一个名为 `instant` 的变量，虽然 `Time` 中的 `instant` 类型与其他两个不同，但三者都是用于某一基础周期的大小。差别在于，`Date` 基于 `Day` 周期；`DateTime` 基于 `Millisecond` 周期；而 `Time` 则基于 `Nanosecond` 周期。

事实上，上例中 `Date` 与 `DateTime` 都会涉及跨天问题，与历史基准时间有关，其中 `UTInstant` 类型的成员变量 `instant` 描述的是一个连续递增的机器时间轴，以世界时^①或格林尼治时间为标准。而且类型 `Date` 与 `DateTime` 的设计遵循 ISO 8601 标准^②，使用的历法是格里高利历^③，也就是我们通常所说的公历。

其中的 `DateTime` 类型本身并不处理时区问题，类似于 Java8 中的 `LocalDateTime` 结构。如果需要，可选用 Julia 的 `TimeZones.jl` 包^④来处理时区。

需要注意的是，ISO 8601 标准对公元前日期的表达有些特别。一般来说，公元前最后一天是“公元前 1 年 12 月 31 日”，其后便是“公元 1 年 1 月 1 日”。但该标准中，公元前 1 年作 0 处理，并采用了负号表示法，即 0000 年 12 月 31 日是公元 0001 年 1 月 1 日的前一天，而 -0001 年表述的是公元前 2 年，-0002 年则表述的是公元前 3 年，以此类推。

11.2 构造

如果要创建一个 `DateTime` 对象，则可通过如下构造方法来实现：

```
DateTime(year, [month, day, hour, minute, second, millisecond]) -> DateTime
```

其中，每个参数的意义正如其英文字面含义。除 `year` 外，其他参数均可省略，但要求提供的每个参数必须能够自动转换到 `Int64` 类型。

下面给出一些构造 `DateTime` 的例子：

```
julia> using Dates
```

- ① 世界时 (Universal Time, UT) 也称格林尼治时间 (Greenwich Mean Time, GMT)，是一种以地球自转为基础的时间计量系统。格林尼治是伦敦原皇家格林尼治天文台所在地，是地球本初子午线的标界，所以作为世界计算时间和经度的起点。UT 有几个版本，最常用的是协调世界时间 (UTC) 和 UT1。UTC 基于国际原子时间，添加闰秒后与 UT1 的误差在 0.9 秒内。
- ② 国际标准 ISO 8601 的文本名称为《数据存储和交换形式·信息交换·日期和时间的表示方法》，它约定了日期和时间的表示方法。
- ③ 格里高利历 (Gregorian Calendar)，由罗马天主教皇格里高利十三世 (Pope Gregory 13) 于公元 1582 年 10 月 15 日启用。格里高利十三世为了精确计算耶稣复活日，在儒略历的基础上，对闰年设置做了调整，建立了该历法。此历法已被广泛采用，成为国际通用历法。我国在公元 1912 年 1 月 1 日启用该历法。
- ④ `TimeZones.jl` 包的代码位置为 <https://github.com/JuliaTime/TimeZones.jl>。其使用的时区数据库由 IANA 提供，地址为 <https://www.iana.org/time-zones>。



```
julia> DateTime(2013)
2013-01-01T00:00:00

julia> DateTime(2013,7)
2013-07-01T00:00:00

julia> DateTime(2013,7,1)
2013-07-01T00:00:00

julia> DateTime(2013,7,1,12)
2013-07-01T12:00:00

julia> DateTime(2013,7,1,12,30)
2013-07-01T12:30:00

julia> DateTime(2013,7,1,12,30,59)
2013-07-01T12:30:59

julia> DateTime(2013,7,1,12,30,59,1)
2013-07-01T12:30:59.001

julia> DateTime(1)      # v1.0新版中不再支持无参调用
0001-01-01T00:00:00
```

在开发中经常需要获得机器的当前最新时间，调用 Dates 中的 now() 函数即可。而该函数返回的对象便是 DateTime 类型，例如：

```
julia> now()
2018-05-21T12:27:26.367

julia> typeof(ans)
DateTime
```

该函数会上报当前的日期，并返回精确到毫秒的时间。

如果要表达更精细的时间，可选用 Time 类型，其构造方法为：

```
Time(hour, [minute, second, millisecond, microsecond, nanosecond]) -> Time
```

可见，其精度能够达到纳秒。

创建 Time 对象类似于创建 DateTime 对象，只需逐一填入参数即可，例如：

```
julia> Time(12, 30, 42, 11, 222, 399)
12:30:42.011222399

julia> Time(1)      # v1.0新版中不再支持无参调用
00:00:00
```

如果应用中不需要表达时间，仅需表达日期，则可采用 Date 类型，其构造方法原型为：

```
Date(year, [month, day]) -> Date
```

该方法非常简单，只有年、月、日三个参数。创建 Date 对象的示例如下：

```
julia> Date(2013)
2013-01-01

julia> Date(2013,7)
```




```
2013-07-01
```

```
julia> Date(2013,7,1)
2013-07-01
```

```
julia> Date(1)           # v1.0新版中不再支持无参调用
0001-01-01
```

对于上述三者描述日期和时间的类型，除了介绍的构造方法外，还可以基于 `Period` 类型的各子类型对象进行构造，原型分别为：

```
DateTime(p::Period...) -> DateTime
Date(p::Period...) -> Date
Time(p::Period...) -> Time
```

其中，参数 `p` 被限定为抽象类型 `Period`，所以其值可以是 `Period` 的任意子类型，例如 `Day` 及 `Year` 等。而且这种构造方法支持可变参数，也没有参数顺序的限制。例如：

```
julia> using Dates
```

```
julia> DateTime(Year(2007), Second(19))
2007-01-01T00:00:19
```

```
julia> DateTime(Month(10), Millisecond(253), Year(2014))
2014-10-01T00:00:00.253
```

```
julia> DateTime(Month(10), Millisecond(253), Year(2014), Hour(9))
2014-10-01T09:00:00.253
```

```
julia> Date(Year(2013), Month(7), Day(1))
2013-07-01
```

```
julia> Date(Month(7), Year(2013))
2013-07-01
```

```
julia> Time(Hour(12), Second(42), Minute(30), Nanosecond(399))
12:30:42.000000399
```

这种方式下，如果一些字段未提供则会取默认值。

另外，`DateTime` 与 `Date` 对象可以进行相互转换，例如：

```
julia> dt = now()
2018-04-16T17:42:35.675
```

```
julia> typeof(dt)
DateTime
```

```
julia> d = Date(dt)
2018-04-16
```

```
julia> dt2 = DateTime(d)
2018-04-16T00:00:00
```

在 `DateTime` 转为 `Date` 类型时，时间部分（包括小时、分钟、秒等）会被丢弃；而在 `Date` 转为 `DateTime` 时，时间值则会被默认置零。



另外, `DateTime` 对象也可以转为 `Time` 类型, 例如:

```
julia> dt = now()  
2018-04-16T18:13:20.744
```

```
julia> t = Time(dt)  
18:13:20.744
```

因为 `DateTime` 精度比 `Time` 低, 所以此时的转换中, 新生成 `Time` 对象中的微秒与纳秒会默认置零。

但是, `Julia` 并不支持 `Time` 向 `DateTime` 的转换, 这点需要特别注意, 例如:

```
julia> DateTime(t)  
ERROR: MethodError: Cannot `convert` an object of type Time to an object of type DateTime
```

11.3 访问

作为 `TimeType` 类型的对象, 从 `Date` 类型、`DateTime` 类型及 `Time` 类型的对象中均可提取各周期的值。以 `Date` 对象为例, 如下:

```
julia> d = Date(2014, 1, 31)  
2014-01-31
```

```
julia> year(d)  
2014
```

```
julia> month(d)  
1
```

```
julia> week(d)  
5
```

```
julia> day(d)  
31
```

注意这些访问函数的首字母是小写的, 而返回的周期值均是整型。

如果要获得 `Period` 类型的周期值对象, 可使用相应函数的首字母大写版, 代码如下:

```
julia> d = Date(2014, 1, 31)  
2014-01-31
```

```
julia> Year(d)  
2014 years
```

```
julia> typeof(ans)  
Base.Dates.Year
```

```
julia> Month(d)  
1 month
```

```
julia> Week(d)
```



```
5 weeks
```

```
julia> Day(d)
31 days
```

为了方便，Julia 还提供一些函数，可以同时获得多个周期的组合值，例如：

```
julia> yearmonth(d)
(2014, 1)
```

```
julia> monthday(d)
(1, 31)
```

```
julia> yearmonthday(d)
(2014, 1, 31)
```

或者：

```
julia> yr, mnth, dy = yearmonthday(d)
(2014, 1, 31)
```

```
julia> yr
2014
```

```
julia> mnth
1
```

```
julia> dy
31
```

甚至，也可以直接访问内部结构中的成员变量值：

```
julia> dump(d)
Date
  instant: Dates.UTInstant{Day}
  periods: Day
  value: Int64 735264
```

```
julia> t.instant
Dates.UTInstant{Day}(735264 days)
```

```
julia> Dates.value(d)
735264
```

其中，value 记录日期 d 距离公元第一天的总天数。

对于 DateTime 结构，同样可以方便地对其中的各周期进行访问：

```
julia> dt = DateTime(2014, 1, 31, 11, 32, 56, 523)
2014-01-31T11:32:56.523
```

```
julia> hour(dt)
11
```

```
julia> minute(dt)
32
```

```
julia> second(dt)
```



```
56
```

```
julia> millisecond(dt)
523
```

并可获得其 `Period` 类型的周期对象:

```
julia> Hour(dt)
11 hours
```

```
julia> Millisecond(dt)
523 milliseconds
```

同样, `Time` 类型也可获得其各周期的值, 例如:

```
julia> t = Time(11,32,56,523,788,699)
11:32:56.523788699
```

```
julia> microsecond(t)
788
```

```
julia> nanosecond(t)
699
```

同样, 也可以获得其各周期的对象:

```
julia> Millisecond(t)
523 milliseconds
```

```
julia> Microsecond(t)
788 microseconds
```

```
julia> Nanosecond(t)
699 nanoseconds
```

可见, Julia 在设计时考虑得非常周全, 为开发者提供了大量方便的接口函数, 能够极为有效地提高开发效率。

11.4 解析

日期 / 时间结构与字符串之间的转换是经常会遇到的, Julia 提供了灵活的转换方式。首先, 我们看看如何将包括 `Date`、`DateTime` 及 `Time` 三者的 `TimeType` 类型内容转换为字符串。

若要将 `TimeType` 类型转为字符串, 可使用 `Dates.format()` 函数实现, 其原型为:

```
format(dt::TimeType, fmt::AbstractString; locale="english") -> AbstractString
```

其中, `fmt` 是一串字符编码, 用于指定日期 / 时间在转换过程中以什么格式输出, 其具体约定如表 11-1 所示。



表 11-1 日期 / 时间转换为字符串的格式规则

格 式 码	示 例	说 明
y	6	年份数值的某位数字 (可限定固定宽度)
Y	1996	最小宽度的年份数值
m	1, 12	月份数值 (最小宽度)
u	Jan	月份英文缩写 (3 个字符)
U	January	月份英文全称
d	1, 31	月份中的某日
H	0, 23	24 小时制的小时数值 (最小宽度)
M	0, 59	分钟数值
S	0, 59	秒数值
s	000, 500	毫秒值 (3 位数字)
e	Mon, Tue	周的某日的英文缩写
E	Monday	周的某日的英文全称

下面给出一些转换的具体例子:

```
julia> Dates.format(now(), "yyyy年mm月dd日 HH:MM:SS")
"2018年04月16日 18:37:19"
```

```
julia> Dates.format(now(), "d u Y,HH:MM:SS.s")
"16 Apr 2018,18:38:48.127"
```

```
julia> Dates.format(now(), "yyyy-mm-dd") # 没有提供时间格式, 时间数据被忽略
"2018-04-16"
```

在转换过程中, 如果格式串没有给出所有周期, 则输出的字符串不会出现该周期, 相应数据也会被忽略。如上例中的最后一个, 没有给出时间部分的输出格式, 所以生成的字符串也不会出现时间的内容。

解析的另一方便便是从字符串中提取日期 / 时间, 并创建 `TimeType` 的对象。这样的过程是通过 `Date` 与 `DateTime` 的构造方法实现的, 原型如下:

```
DateTime(dtstr::AbstractString, format::AbstractString; locale="english") -> DateTime
Date(dtstr::AbstractString, format::AbstractString; locale="english") -> Date
```

其中, `dtstr` 是包含日期 / 时间的字符串; `format` 用于描述 `dtstr` 的格式。

需要了解的是, 在 Julia 中并没有提供字符串解析为 `Time` 对象的功能, 但可以通过 `DateTime` 类型进行中转, 即先将字符串转为 `DateTime` 对象, 然后再将该对象转为 `Time` 类型。这种转换过程中同样需要进行格式说明, 不过与表 11-1 中所列的对象转字符串的规则会有些差异。

表 11-2 字符串转日期 / 时间的格式规则

格 式 码	匹 配 示 例	说 明
y	1996, 96	返回 1996, 0096
Y	1996, 96	返回 1996, 0096, 等效于 y
m	1, 01	匹配 1 或 2 个数值的月份
u	Jan	匹配月份的英文缩写
U	January	匹配月份的英文全称
d	1, 01	匹配 1 或 2 个数值的天
H	00	匹配小时
M	00	匹配分钟
S	00	匹配秒
s	.500	匹配毫秒
e	Mon, Tues	匹配周的某天 (英文缩写)
E	Monday	匹配周的某天 (英文全称)
yyymmdd	19960101	匹配固定宽度的年、月、日

下面给出一些将字符串解析为 Date 或 DateTime 对象的例子, 如下:

```
julia> dt1 = Date("2015-01-01", "y-m-d")
2015-01-01

julia> typeof(dt1)
Date

julia> dt2 = DateTime("20150101", "yyymmdd")
2015-01-01T00:00:00

julia> typeof(dt2)
DateTime

julia> dt3 = Date("2015年3月13日", "y年m月d日")
2015-03-13

julia> typeof(dt3)
Date
```

需要注意的是, 在将字符串转为指定日期 / 时间格式并生成对象时, 会有一些性能损耗。为此, 最好能预先利用 DateFormat 结构对描述格式化方法的字符串内容进行封装, Julia 会在此基础上提供优化能力。这样, 新的基于字符串解析的构造方法原型为:

```
DateTime(dtstr::AbstractString, df::DateFormat) -> DateTime
Date(dtstr::AbstractString, df::DateFormat) -> Date
```

其中, DateFormat 类型的参数 df 即为预先定义的格式描述对象。

DateFormat 对象的创建有两种方式: 一是将格式字符串作为参数调用其构造方法;

二是直接在某字符串前以 `dateformat` 为前缀标识。例如：

```
julia> DateFormat("y-m-d")
dateformat"y-m-d"

julia> dateformat"y-m-d HH:MM:SS"
dateformat"y-m-d HH:MM:SS"
```

在开发中选择其中一种便可。但如果在循环中提供 `DateFormat` 对象，最好选用后者，因为它是一种宏，在首次调用时会展开，只会进行一次实例化。

下面看几个使用 `DateFormat` 进行解析的例子：

```
julia> df = DateFormat("y-m-d");

julia> dt = Date("2015-01-01",df)
2015-01-01

julia> dt2 = Date("2015-01-02",df)
2015-01-02
```

或者：

```
julia> df = dateformat"y-m-d HH:MM:SS"
dateformat"y-m-d HH:MM:SS"

julia> dt = DateTime("2017-11-21 13:45:32", df)
2017-11-21T13:45:32
```

至于为何建议选择 `DateFormat` 方式而不是字符串方式提供解析格式的描述信息，可以通过大规模循环的例子对比两种方式的差异，如下：

```
julia> @time for i = 1:10^5
           Date("2015-01-01", dateformat"y-m-d") # DateFormat方式
       end
0.005087 seconds (100.00 k allocations: 3.052 MiB)

julia> @time for i = 1:10^5
           Date("2015-01-01", "y-m-d") # 字符串方式
       end
2.733261 seconds (9.60 M allocations: 395.203 MiB, 6.27% gc time)
```

从上例可见，无论是在内存占用方面还是耗时方面，两者都表现出了巨大的差距，甚至有数百倍的性能差异。所以，在将字符串解析为日期对象时，使用 `DateFormat` 方式应该是最佳选择。

11.5 运算

11.5.1 早晚比较

显然，无论是日期还是时间都是有序的对象，所以它们之间能够进行大小的比较。例如 `Date` 类型的对象之间做比较：

```
julia> d1 = Date(2014,1,31);
julia> d2 = Date(2015,2,12);

julia> d1 < d2
true

julia> d1 >= d2
false

julia> Date(2015, 2, 12) === d2
true
```

或者:

```
julia> dt1 = DateTime(2015, 3, 9, 12, 38, 9, 133)
2015-03-09T12:38:09.133

julia> dt2 = DateTime(2017, 4, 9, 11, 28, 9, 253)
2017-04-09T11:28:09.253

julia> dt1 > dt2
false

julia> dt1 !== dt2
true
```

当然, Time 之间也是可以比较的:

```
julia> t1 = Time(14, 28, 37, 12,800,970)
14:28:37.01280097

julia> t2 = Time(15, 28, 37, 12,800,970)
15:28:37.01280097

julia> t1 < t2
true

julia> t1 === t2
false
```

另外, Date 类型可以与 DateTime 类型之间进行比较, 例如:

```
julia> dt1 > d2      # 即DateTime(2015, 3, 9, 12, 38, 9, 133) > Date(2015, 2, 12)
true

julia> dt3 = DateTime(2015, 2, 12)
2015-02-12T00:00:00

julia> d2 == dt3     # d2的内容为2015-02-12
true
```

尤其是其中最后的示例, 虽然两者类型不同, 但因为时间默认为 0 而日期一致, 所以判断是否相等时成立。

但 Time 对象不能与 Date 或 DateTime 对象进行比较, 例如:

```
julia> t1 > d2
```

```
ERROR: promotion of types Date and Time failed to change any arguments
```

```
julia> t1 < dt1
```

```
ERROR: promotion of types Time and DateTime failed to change any arguments
```

这是因为 Time 并不涉及历史时间，内部成员的类型与其他两者的也并不相同，从上文中它们的内部结构便能够看出来。

11.5.2 时长计算

对于日期或时间，计算两者之间的时长跨度是经常遇到的问题，我们只需将两个对象进行减运算便可获得这样的结果，例如：

```
julia> d1 - d2           # 即Date(2014, 1, 31) - Date(2015, 2, 12)
-377 days
```

```
julia> typeof(ans)
Day
```

```
# 即DateTime(2017, 4, 9, 11, 28, 9, 253) - DateTime(2015, 3, 9, 12, 38, 9, 133)
julia> dt2 - dt1
65832600120 milliseconds
```

```
julia> typeof(ans)
Millisecond
```

```
# 即Dates.Time(15, 28, 37, 12,800,970) - Dates.Time(14, 28, 37, 12,800,970)
julia> t2 - t1
3600000000000 nanoseconds
```

```
julia> typeof(ans)
Nanosecond
```

可见，Date 之间计算时长时，返回的是相差的 Day 类型，表示它们之间相隔的天数；而 DateTime 之差返回的是 Millisecond 对象，表示两者间相差的毫秒数；Time 之差则是 Nanosecond 对象，其值为两者间相差的纳秒数。这是以各自类型的最高精度类型来表示对象间的时长间隔，其实与 TimeType 三个类型表达时间的机制有关。

但是这种计算时长的运算不像上述的大小比较运算（能够跨 Date 与 DateTime 类型），只能在同类型间进行。而且对于 TimeType 类型而言，加法、除法及乘法等其他算术运算也是不支持的，但可以与 Period 对象进行加减运算，从而达到对时间或日期进行调整的目的，例如：

```
julia> Date(2014,1,3) - Day(2) + Month(3) - Year(3)
2011-04-01
```

```
julia> DateTime(2014,1,3,10,31,15,378) - Millisecond(200) + Hour(3) - Minute(17) +
Second(20)
2014-01-03T13:14:35.178
```

```
julia> Time(10,9,18,10,20,30) + Nanosecond(300) - Hour(2) - Minute(40)
07:29:18.01002033
```


这种运算不但能够链式进行，而且没有顺序要求，只需让周期修改值以 Period 对象的方式参与计算即可，极为方便。

实际上，Period 类型本身的意义便表达了某周期精度下的时间跨度，本质上是一种数值，不像 TimeType 三个类型那样表达的是时间点，有着严格的物理意义，所以在运算上要更灵活些。例如，Period 类型的对象之间便可进行加减运算：

```
julia> Year(2017) + Year(10)
2027 years
```

```
julia> typeof(ans)
Year
```

```
julia> Second(35) - Second(20)
15 seconds
```

```
julia> typeof(ans)
Second
```

Period 类型不但可以像上例中进行同类型计算，还可以跨类型计算，例如：

```
julia> Month(5) + Day(2)
5 months, 2 days
```

```
julia> typeof(ans)
Dates.CompoundPeriod
```

```
julia> Millisecond(42) - Microsecond(30)
42 milliseconds, -30 microseconds
```

```
julia> typeof(ans)
Dates.CompoundPeriod
```

差别在于计算结果的类型不同。其中，Dates.CompoundPeriod 类型其实是一个元素类型为 Period 的一维数组，记录着各周期类型的对应值。

对于 Period 类型的对象，还支持另外一种运算，即求余数，例如：

```
julia> Month(5) % Month(2)
1 month          # 整除后余1
```

```
julia> typeof(ans)
Month
```

```
julia> Millisecond(35) % Millisecond(3)
2 millisecond      # 整除后余2
```

```
julia> typeof(ans)
Millisecond
```

Julia 支持这种操作是有道理的，因为该运算通常会应用于需要时间对齐的场景中。

乘法对于 Period 类型是无意义的，所以并不支持，但除法在操作数均为同类型的情况下是可以进行的，只不过从某种意义上讲，只是作为普通的数值进行计算，例如：

```
julia> Day(10) / Day(3)
```

```
3.3333333333333335
```

```
julia> typeof(ans)
Float64
```

支持这样的运算，主要是为了在一些涉及数值计算的场景中更为方便。

11.5.3 时间序列

以 `Period` 类型的加减运算及能够对 `TimeType` 对象进行加减调整的功能为基础，我们可以在范围表达式中使用日期/时间，从而构造出迭代器，并可生成时间序列。例如，可以生成两个日期之间的其他日期，如下：

```
julia> dr1 = Date(2014,2,26) : Day(1) : Date(2014,3,2)
2014-02-26:1 day:2014-03-02
```

```
julia> for d in dr1
    println(d)
end
2014-02-26
2014-02-27
2014-02-28
2014-03-01
2014-03-02
```

```
julia> collect(dr1)
5-element Array{Date,1}:
 2014-02-26
 2014-02-27
 2014-02-28
 2014-03-01
 2014-03-02
```

而且这种方式可以自动实现正常的跨月处理（2014 年 2 月只有 28 天）。需要注意的是，在 `v1.0` 版中，日期时间的这种序列生成方法不再像之前的版本允许省略中间的 `step` 参数，所以中间的时间步长参数是必需的。

另外，在步长的设置中，可以选用 `Period` 中的任意一个子类型对象作为参数，例如：

```
julia> dr3 = DateTime(2014,2,26) : Second(45) : DateTime(2014,2,26,0,9,30)
2014-02-26T00:00:00:45 seconds:2014-02-26T00:09:00
```

```
julia> collect(dr3)
13-element Array{DateTime,1}:
 2014-02-26T00:00:00
 2014-02-26T00:00:45
 2014-02-26T00:01:30
 2014-02-26T00:02:15
 2014-02-26T00:03:00
 2014-02-26T00:03:45
 2014-02-26T00:04:30
 2014-02-26T00:05:15
 2014-02-26T00:06:00
 2014-02-26T00:06:45
```

```
2014-02-26T00:07:30
2014-02-26T00:08:15
2014-02-26T00:09:00
```

其中，以 `DateTime` 类型的两个对象作为起止时间点，将 `Dates.Second` 对象作为步长因子，得到了相邻时间间隔为 45 秒的时间序列。

当然这种以时间为基础的范围表达式也可以采用 `Time` 类型，其中的步长可以使用毫秒、纳秒等。但若以 `Day` 等跨天的周期作为 `Time` 类型时间点的步长，会报错，例如：

```
julia> dr4 = Time(12,30,21,10,20,30,40) : Day(1) : Time(23,30,35,20,30,40,50)
ERROR: MethodError: no method matching Time{::Int64, ::Int64, ::Int64, ::Int64, ::Int64, ::Int64, ::Int64}
```

其实这个道理类似于 `Time` 不能与 `Date`、`DateTime` 混合比较，因为它们内部的机制有所不同，而且也没有实际的意义。

11.5.4 周期舍入

在时间序列类数据的处理中，常常需要将记录的时间戳对齐到某个周期的时间框架中。这期间便需要将各种不同的时间进行转换，并找到周期框架内的标准时间。典型的场景是：金融行情系统中，国内实时的行情数据一般会一秒两次地进行传输（所谓 `Tick` 数据），每一条记录除了最新价等价格数据外，最为重要的信息便是记录生成的时间，一般精度到毫秒级，但每次的毫秒值并不全是一致的。例如有一些数据如表 11-3 所示。

表 11-3 金融 `Tick` 数据片段

时 间	最新价	成交量	时 间	最新价	成交量
2018-02-18 14:12:38.495	32.5	10632	2018-02-18 14:12:40.102	34.5	16387
2018-02-18 14:12:38.998	31.8	11254	2018-02-18 14:12:40.511	33.7	15916
2018-02-18 14:12:39.001	33.1	9503	2018-02-18 14:12:40.997	30.9	10231
2018-02-18 14:12:39.503	34.2	8769			

在这些数据接收后，假设需要生成 1 秒周期的数据，则这种周期中的开盘价、收盘价、最高价及最低价会依据 1 秒周期内各项记录中的最新价进行生成，结果应类似于表 11-4 所示。

表 11-4 舍入到秒级周期数据的结果数据

时 间	开 盘 价	收 盘 价	最 高 价	最 低 价	成 交 量
2018-02-18 14:12:38	32.5	32.5	32.5	32.5	10632
2018-02-18 14:12:39	31.8	33.1	33.1	31.8	20757
2018-02-18 14:12:40	34.2	34.5	34.5	34.2	25156
2018-02-18 14:12:41	33.7	30.9	33.7	30.9	26147

暂不论怎么得到表中后五项的价格与成交量数据，只研究时间的规整问题。由于毫秒值的不确定（仅能确定应在 0 毫秒与 500 毫秒周边波动），显然直接将原始数据的毫秒值向以秒

为周期的时间框架对齐是不合适的，至少要按照毫秒值进行四舍五入后再合并归约。

针对这类时间处理场景，Julia 提供了方便的 `round()` 函数，能够按照指定的周期对给定的时间进行转换，将其对齐到指定周期的时间框架中。我们仍以上述的时间序列为例，为了方便，将时间放入一个数集中，代码如下：

```
julia> using Dates

julia> fmt = dateformat"y-m-d HH:MM:SS.s"

# 原始数据中的时间序列
julia> tms = ("2018-02-18 14:12:38.495",
              "2018-02-18 14:12:38.998",
              "2018-02-18 14:12:39.001",
              "2018-02-18 14:12:39.503",
              "2018-02-18 14:12:40.102",
              "2018-02-18 14:12:40.511",
              "2018-02-18 14:12:40.997")

# 调用round()函数将上述的时间字符串转为DateTime结构
julia> rtms = [round(x, Second(1)) for x in tms]
7-element Array{DateTime,1}:
2018-02-18T14:12:38
2018-02-18T14:12:39
2018-02-18T14:12:39
2018-02-18T14:12:40
2018-02-18T14:12:40
2018-02-18T14:12:41
2018-02-18T14:12:41

# 对DateTime序列进行舍入处理
julia> rtms = [(x, round(x, Second(1))) for x in tms]
7-element Array{Tuple{DateTime,DateTime},1}:
(2018-02-18T14:12:38.495, 2018-02-18T14:12:38) # 结果以Tuple结果列出，便于对比
(2018-02-18T14:12:38.998, 2018-02-18T14:12:39) # 左边是原始时间，右边是舍入后的时间
(2018-02-18T14:12:39.001, 2018-02-18T14:12:39)
(2018-02-18T14:12:39.503, 2018-02-18T14:12:40)
(2018-02-18T14:12:40.102, 2018-02-18T14:12:40)
(2018-02-18T14:12:40.511, 2018-02-18T14:12:41)
(2018-02-18T14:12:40.997, 2018-02-18T14:12:41)
```

如此处理后，结果中的时间便均会舍去不需要的小周期，在秒值上能够被指定的周期值整除。我们再看另外一个例子：

```
julia> dr4 = DateTime(2014,2,26,9,0,0) : Second(1) : DateTime(2014,2,26,9,0,30)
2014-02-26T09:00:00:1 second:2014-02-26T09:00:30

julia> [(x, round(x, Second(3))) for x in dr4]
31-element Array{Tuple{DateTime,DateTime},1}:
(2014-02-26T09:00:00, 2014-02-26T09:00:00)
(2014-02-26T09:00:01, 2014-02-26T09:00:00)
(2014-02-26T09:00:02, 2014-02-26T09:00:03)
(2014-02-26T09:00:03, 2014-02-26T09:00:03)
(2014-02-26T09:00:04, 2014-02-26T09:00:03)
(2014-02-26T09:00:05, 2014-02-26T09:00:06)
```



```
(2014-02-26T09:00:06, 2014-02-26T09:00:06)
(2014-02-26T09:00:07, 2014-02-26T09:00:06)
(2014-02-26T09:00:08, 2014-02-26T09:00:09)
(2014-02-26T09:00:09, 2014-02-26T09:00:09)
(2014-02-26T09:00:10, 2014-02-26T09:00:09)
(2014-02-26T09:00:11, 2014-02-26T09:00:12)
(2014-02-26T09:00:12, 2014-02-26T09:00:12)
:
(2014-02-26T09:00:19, 2014-02-26T09:00:18)
(2014-02-26T09:00:20, 2014-02-26T09:00:21)
(2014-02-26T09:00:21, 2014-02-26T09:00:21)
(2014-02-26T09:00:22, 2014-02-26T09:00:21)
(2014-02-26T09:00:23, 2014-02-26T09:00:24)
(2014-02-26T09:00:24, 2014-02-26T09:00:24)
(2014-02-26T09:00:25, 2014-02-26T09:00:24)
(2014-02-26T09:00:26, 2014-02-26T09:00:27)
(2014-02-26T09:00:27, 2014-02-26T09:00:27)
(2014-02-26T09:00:28, 2014-02-26T09:00:27)
(2014-02-26T09:00:29, 2014-02-26T09:00:30)
(2014-02-26T09:00:30, 2014-02-26T09:00:30)
```

在上例中, 为了方便理解, 目标时间框架中的对齐点发生变化时将字体加粗处理。稍作分析便能够发现, 原始时间转换时, 会向离得最近的目标时间对齐。

再看另外一个例子:

```
julia> round(DateTime(2016, 7, 17, 11, 55), Hour(10))
2016-07-17T12:00:00
```

这是一个特别的例子, 得到的结果时间中虽然分钟和秒等更小周期的时间已经被舍弃, 但在小时值 12 并不是 10 的倍数。原因在于: 2016-07-17T12:00:00 对应的时间戳 (Timestamp) 是相距初始起点 0000-01-01T00:00:00 之后的 17676660 个小时, 而这个值是能被 10 整除的。

Julia 中 Date 及 DateTime 对象的表达均遵循 ISO 8601 标准, 而周期舍入计算中的舍入元时 (Rounding Epoch) 选择的基准时间是 0000-01-01T00:00:00, 即公元前 1 年 1 月 1 日 0 时。这所有的舍入计算中, 所依据的均是距此舍入元时的时长值 (天、时分秒、毫秒等), 而间隔时长的单位自然与所指定的周期一致。之所以该舍入元时选择的并非是公元前最后一天 (0000-12-31T00:00:00), 为的便是在舍入计算中能够避免发生更多的混淆。不过对于该舍入元时, 在对周这个周期进行计算时有些特别: 四舍五入到最近的周时间点时, 总会返回星期一, 即 ISO 8601 约定的每周第一天, 所以当对周进行舍入计算时, 会采用另外一个舍入元时, 即 0000-01-03T00:00:00, 这是 ISO 8601 约定的公元前 1 年的第一周的第一天。

接着看一个月份的例子:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Month(2))
2016-07-01T00:00:00
```

结果仍不是 2 的倍数, 而且天是 1 号。问题的关键在于每月的首日均是 1 号 (1-Indexed), 并不像小时、分钟、秒、毫秒等是 0-Indexed, 即首数是 0。关于这方面的更多内容, 可参考 Dates 模块的 API 文档。

11.6 属性

对于 `TimeType` 类型的对象，还提供了一些属性特征的判断函数，例如：

```
julia> d = Date(2014, 8, 16)
2014-08-16

julia> monthname(d)                # 获得月份的英文全称
"August"

julia> isleapyear(d)                # 判断年份是否是闰年
false

julia> dayofyear(d)                 # 该日期是当年的第几天
228

julia> quarterofyear(d)             # 该日属于第几季度
3

julia> dayofquarter(d)              # 该日是所属季度的第几天
47
```

而且，能够以某日期为依据获得某些特定的日期，例如：

```
# 获得当前日期所属周的第一天（注意，未必是周一）
julia> firstdayofweek(Date(2014,7,16))
2014-07-14

julia> lastdayofmonth(Date(2014,7,16))    # 获得当前日期所属月的最后一天
2014-07-31

julia> lastdayofquarter(Date(2014,7,16))  # 获得当前日期所属季度的最后一天
2014-09-30
```

上例其实以某 `TimeType` 为起点，沿着时间轴后推或回溯，搜索某些属性特别的时间。对于这种需求场景，Julia 专门设计了两个函数用于实现这样的功能。这两个函数的基本原型为：

```
toprev(func::Function, dt::TimeType; step=Day(-1), limit=10000, same=false) -> TimeType
tonext(func::Function, dt::TimeType; step=Day(1), limit=10000, same=false) -> TimeType
```

其中，函数对象 `func` 用于实现自定义的条件，接收 `TimeType` 类型的参数并经过条件判断后返回布尔值；`dt` 是起点时间；`step` 是搜索步长；`limit` 参数用于限定搜索的最多次数（迭代次数）；`same` 参数则指示是否考虑 `dt` 本身。

这两个函数会以 `dt` 为起点、以 `step` 为步长自动向前或向后移动，逐一判断某个 `TimeType` 对象是否满足 `func` 为 `true` 的条件，一旦遇到便会终止（只找一个），若超过 `limit` 次移动后仍未发现则退出，不再继续搜索，并上报错误：

```
ERROR: ArgumentError: Adjustment limit reached: 10000 iterations
```

例如，我们要寻找某个日期之后的第一个周二，获得其日期值，则可如下实现：

```
julia> istuesday = x-> dayofweek(x) == Tuesday    # 如果日期是周二，则返回true
(::#1) (generic function with 1 method)
```

```
julia> tonext(istuesday, Date(2014,7,13))      # 日期2014-07-13实际是周日，故其之后
2014-07-15                                     # 的第三天是周二
```

当然该例子有个更为简洁的实现方式，如下：

```
julia> tonext(Date(2014,7,13), Tuesday)        # 另外一种便捷的使用方式
2014-07-15
```

再例如，我们要找一个时间之前分钟值是 6 的倍数的第一个时间：

```
julia> mts_six_times = x-> Minute(x) % 6 == Second(0) # 前面求余计算的结果是Second
                                                    # 类型
(::#3) (generic function with 1 method)

julia> toprev(mts_six_times, Time(13,2,32,10,20,30); step = Second(10))
13:06:02.01002003
```

可见，这两个函数同样适用于 Time 等类型。

另外，如果条件函数过长，可以借用 do 代码块，使得代码更为清晰。例如，要找到 2014 年 12 月之后的首个感恩节，代码如下：

```
julia> tonext(Date(2014,12,13)) do x
# 如果日期是11月份的第4个周四（感恩节），则返回true
    dayofweek(x) == Thursday && dayofweekofmonth(x)
    == 4 && month(x) == November
end
2015-11-26 # 搜索到的结果，即下一个感恩节是2015年11月26日
```

由此可见，Julia 在日期与时间方面的支持非常全面且强大，能够非常简洁、方便地实现各种功能，这也是软件工程长期实践的结晶。

流 与 IO

数据是程序操作的基础对象，往往被封装在某种固定的数据结构中。但有些数据源会源源不断地产生数据，或者并不是一次性地提供完整的数据，例如串口、并口、USB 口、网口、磁盘等。这样的数据在一段时间内往往无法判断何时结束，例如，文件的字符序列，音频字节流，视频压缩流和网络字节流等，也就难以用某种固定结构的容器将数据一次性地存取。

一般而言，这种连续不断的字节流会按数据源设备提供的输入与输出功能，被分为输入流与输出流，对应着不同的操作特性。通常来说，输入流只支持读操作，而输出流只支持写操作。在 Julia 中，各种流的父类型均是 IO 类型，都从其继承了读 `read()` 和写 `write()` 这两种最基本的操作接口。其中的标准流、文件流和网络流是本章要介绍的重要内容。

12.1 标准流

键盘和显示器分别是最常见的输入和输出设备，已经有了相当成熟的信号编码机制，遵循业界标准。因为它们在交互过程中会不断产生数据流，所以针对性地定义了两类流：标准输入流（standard input stream, `stdin`）和标准输出流（standard output stream, `stdout`）。又因为程序输出信息与系统错误信息有所区别，产生的源头与成因也所有不同，为了区分，将其中的错误输出定义为标准错误流（standard error stream, `stderr`）。

Julia 定义了三个标准流对象，即 `stdin`、`stdout` 和 `stderr`，分别对应标准输入流、输出流及错误流。因为这三者都对应着不同的设备，而这些设备是固定不变的、始终存在并唯一，所以这三者都被定义为全局常量。

三个流对象的类型均为 `Base.TTY`，继承树为：

```
Base.TTY <: Base.LibuvStream <: IO <: Any
```


其中, TTY 是 Teletypes 的一种缩写, 原指电传打字机, 在 Linux 等系统中一般指代虚拟控制台、串口等终端设备; 而 Libuv[⊖]则是 Julia 引入的异步 IO 库, 这不作过多介绍, 读者可参考相关资料; 再之便是 IO 类型, 可以认为是它们的顶层父类型。

作为 IO 类型的子类型常量, 标准流对象也继承了 read() 和 write() 的接口。但 stdin 只能使用 read() 接口函数, 而 stdout 与 stderr 这两个输出流则只可使用 write() 函数。如果要查询 IO 对象在读写方面的适用性或属性状态, 可以通过表 12-1 所示的函数进行。同时, 在该表中还列出了三个标准流的属性。从中可见, stdin 是只读的, 不支持写操作; stdout 与 stderr 则可写不可读。

表 12-1 设备状态查询函数及标准流属性

函 数 名	功 能	stdout	stderr	stdin
isopen()	对象是否打开中, 能进行读或写	true	true	true
isreadable()	IO 对象是否可读	false	false	false
isreadonly()	流对象是否只能读	false	false	true
iswritable()	IO 对象是否可写	true	true	false

1. 输出流

如果对 stdout 调用 write() 函数, 写入的内容其实就是将内容打印到屏幕中, 例如:

```
julia> write(stdout, "Hello World\n", "Sammy\n")
Hello World
Sammy
18
```

其中, 除了打印给定的两个字符串外, 还输出了数字 18, 这其实是 write() 函数的返回值, 表示成功写入的字符数 (REPL 中可在语句后加分号屏蔽)。同为输出流的 stderr 类似:

```
julia> write(stderr, "There is an error!\n"); # 使用分号屏蔽了返回值
There is an error!
```

不过, 函数 write() 不会对内容进行文本转换, 而是原样写入, 例如:

```
julia> write(stdout, 0x61);
a
```

其中, 0x61 是字符 a 的 ASCII 码, 屏幕会直接输出该字符。而若用 print() 函数, 输出为:

```
julia> print(stdout, 0x61)
97
```

可见效果完全不同, 0x61 被作为十六进制数值处理, 并打印该值的十进制值。

事实上, 无论是 write() 还是 read() 操作, 都是基于二进制字节流的, 所以任何数据都会在执行时被处理为 UInt8 序列, 即 Array{UInt8,1} 类型 (见 9.6 节)。例如往 stdout 中写入多字节字符 Unicode 码值:

⊖ Libuv 是一种提供底层异步 IO 操作的跨平台 API, 以事件循环为核心, 基于多进程、多线程、信号等机制, 支持 TCP/UDP 网络交互、文件系统操作与事件、TTY 字符设备、IPC 通信、命名管道等。



```
julia> write(stdout, '\u2200')
∀3
```

其中故意没有屏蔽 `write()` 的返回值，可见 Unicode 码值被处理为 3 字节序列。

2. 输入流

与 `write()` 略有不同的是，`read()` 函数在调用时能指定欲读取数据的类型。例如：

```
julia> read(stdin, Char)
a
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> read(stdin, Char)
123
'1': ASCII/Unicode U+0031 (category Nd: Number, decimal digit)
```

在 REPL 中执行 `read()` 时，回车后光标便会移动到新空白行等待用户输入；当输入字符并回车后，该函数会返回接收到的字符。

如果要读取多字符或字符序列，可以使用 `readline()` 函数；而且该函数能够将获得的字符串返回，例如：

```
julia> a = readline(stdin)
abcdef          # 输入的字符串
```

输入字符串 `abcdef` 回车后，便会反馈以下的字符串对象：

```
"abcdef"
```

其中变量 `a` 会取得该输入内容。

如果要持续不断地逐行读取内容，可结合 `for` 循环使用 `eachline()` 函数，例如：

```
julia> for line in eachline(stdin)
    print("accepted: $line\n")
end
line 1          # 输入的内容
accepted: line 1 # 反馈接收到的内容
line 2
accepted: line 2
```

上述的 `readline()` 函数会在回车时结束输入状态并返回，如果要读取指定数量的序列，可以通过传入数值参数调用 `read()` 函数。例如：

```
julia> b = read(stdin, 4)
abcdef
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

需要注意的是，该函数所限定的数量实际是字节数，即要返回的 `Array{UInt8,1}` 类型对象的阶数，而且多余的输入内容会被舍弃。

当然，也可以在采用该读取方式将接收到的内容填充到已有的数据区内。例如：

```
julia> x = zeros(UInt8, 4)
```





```
4-element Array{UInt8,1}:  
 0x00  
 0x00  
 0x00  
 0x00
```

```
julia> read!(stdin, x)  
abcd  
4-element Array{UInt8,1}:  
 0x61  
 0x62  
 0x63  
 0x64
```

其中, `x` 初始化了 4 个字节空间, 输入四个字符回车后, 数组 `x` 便存储了这些字符 (多余的输入仍会被舍弃)。

如果是限定数量地持续逐字节地读取, 则可采用如下方式:

```
while !eof(stdin)  
    x = read(stdin, Char)  
    println("accepted: $x \n")  
end
```

其中, `eof()` 函数用于判断流内是否仍有内容可读, 此内容会在后文介绍。需注意的是, 因为 `stdin` 总是处于开放状态, 没有结束的时候, 所以该示例是个无限循环。

12.2 文件操作

长期保留或需要转移的数据往往会以文件的方式存储在外部介质中。对于较大的文件, 逐行或逐字节对其进行读取时, 便是一种流处理过程; 写入过程类同。不过与标准流不同的是, 文件流并不是常开的状态, 能在需要的时候进行关闭和打开。所以文件流对象除了继承自 `IO` 类型的 `read()` 和 `write()` 操作外, 还有打开和关闭操作。

1. 打开操作

打开文件的方法原型主要有两种, 如下:

```
open(filename::AbstractString [, [read::Bool, write::Bool, create::Bool,  
    truncate::Bool, append::Bool]])  
open(filename::AbstractString [, mode::AbstractString])
```

执行后, 这两个方法会加载 `filename` 指定的文件, 成功后返回 `IOStream` 类型的对象; 开发中只需根据情况选择其中一种方法即可。

函数中的其他参数用于控制打开的文件对象的操作权限。上述的两种方法分别提供了五元数组与字符串模式的两种方式, 能够在读、些、新建、清空、追加等几个方面进行独立的配置。前种方式只需将对应位置的元素置为 `true` 即可, 后一种方式则可按照表 12-2 所列的格式进行组合设置。当然, 权限配置是可选的, 不设置时默认为只读。

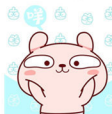




表 12-2 文件操作模式

模 式	操 作	说 明
r	读	只读
r+	读, 写	可读可写
w	写, 新建, 清空	文件不存在则新建, 存在则清空再写入
w+	读、写、新建、清空	可读可写, 不存在就新建, 存在就清空
a	写、新建、追加	只写, 不存在就新建, 存在则追加
a+	读、写、新建、追加	可读可写, 不存在则新建, 存在则追加

例如, 有一个名为 `hello.txt` 的文本文件, 以默认参数打开:

```
julia> f = open("hello.txt")
IOStream(<file hello.txt>)
```

查询获得的 `IOStream` 对象的状态, 如下:

```
julia> isopen(f), isreadable(f), isreadonly(f), iswritable(f)
(true, true, true, false)
```

可见, 该文件流对象 `f` 处于打开状态, 但可读不可写。

2. 读取操作

此时, 可以使用 `readlines()` 函数读取其全部的文本内容, 获得类似如下的内容:

```
julia> readlines(f)
4-element Array{String,1}:
 "the content of the file hello.txt"
 "this is the second line."
 ""
 "the forth line."
# 各行内容会分别放入String类型的向量中
# 空行
```

该函数会将读取的内容按行放入 `String` 向量中, 空行会对应空字符串。

当然对于内容很大的文本文件, 可以通过 `readline()` 函数逐行读取, 例如:

```
julia> while !eof(f)
    println(readline(f))
end
the content of the file hello.txt
this is the second line.

the forth line.
```

如前文所述, 该函数会将单行返回为 `String` 类型的对象, 然后对其进行各种处理。

类似于对 `stdin` 的读取, 也可以通过 `read()` 按类型逐元素读取文件流, 或以字节流的格式读取指定字节数的 `Array{UInt8,1}` 对象。若要将数据填入指定大小的数组缓存区, 除了函数 `read!()` 外, 还可以选择 `readbytes!()` 函数。

`readbytes!()` 函数在将内容读入时, 多出预先分配空间的数据不会被舍弃; 如果指定读入及可读入的字节数大于给定数组的大小, 该函数会自动放大数组, 以容纳足够的内容。例如:





```
julia> data = zeros{UInt8, 2}
2-element Array{UInt8,1}:
 0x00
 0x00

julia> readbytes!(f, data, 3)
3

julia> data
3-element Array{UInt8,1}:
 0x20
 0x63
 0x6f
```

其中, 缓存数组 `data` 大小为 2, 但要求读取 3 字节, 执行后可见 `data` 被放大到了 3 字节, 并被填入了有效的数据。同时, 该函数会在执行成功后上报已读入的字节数。

如果反复地读取文件数据, 会发现之后的读取操作再没有获得更多的内容。这是因为内部存在一个流控制的游标, 此时已经移到了文件尾部。若要再次读取到数据, 需要重置该游标或将其移动到规定的位置。关于这方面的内容会在后文详细介绍。

3. 写入操作

对于前面打开的文件流对象 `f`, 如果要直接进行写操作, 会发现:

```
julia> write(f, "Hello again.")
ERROR: ArgumentError: write failed, IOStream is not writeable
```

提示了 “write failed” 错误, 并告知该对象不可写。这是因为打开时采用的默认权限配置, 所以 `f` 是只读的。

为此, 可通过 `close()` 函数先将 `f` 关闭, 再以新的权限模式打开该上述文件, 即:

```
julia> f = open("hello.txt", "a") # 以追加方式打开
IOStream(<file hello.txt>)
```

随后, 便可往流对象 `f` 中写入内容了:

```
julia> write(f, "new line.")
9
```

之后便能够发现原文件中多了上例中写入的内容。

当然, 除了字符串外, 也可以将二进制字节流写入文件中。例如:

```
julia> x = UInt8['a', 'b', 'c', 12, 39]
5-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x0c
 0x27

julia> write(f, x)
5
```

可见, `x` 中的 5 个字节都被成功写入文件对象 `f` 中。





除此之外，写入操作还支持更多类型，此处不再赘述。需要注意的是，写入文件会对文件的内容进行更改，甚至会预先清空之前的内容，所以在对文件流对象进行写操作时，一定要小心。而且，在写操作完成后，建议习惯性地关闭文件流对象，从而释放被占据的各种资源，并确保新增的内容及时 flush 到目标流中。

4. do 操作

事实上，打开文件的方法除了上文介绍的两种之外还有第三种，即：

```
open(f::Function, filename[, mode])
```

其中，函数对象 `f` 提供了文件对象处理的功能。

基于此函数，再结合 `do` 代码块的特点，便可实现文件处理后自动进行清理工作的功能，无须再显式地 `close()` 打开的文件对象。例如：

```
julia> open("hello.txt", "r+") do f          # 打开文件，流对象绑定到变量f
                                          # 各种读或写等操作
end
```

在这种方式中，`do` 代码块执行完之后内部会自动关闭打开的文件流对象，我们只需将注意力集中在 `do` 内部的处理逻辑上，无须自己显式地管理资源。而且即使在 `do` 内部出现异常时，这种自动清理工作也能够顺利执行。

12.3 读写缓存

数据的流动是转化变换的基础，但硬件架构的不同部件设备在传输与处理速度方面存在着各种差异，使得它们之间需要必要的缓冲区（Cache）或缓存（Buffer），以协调各个处理节点间的步调不一致的问题。

Julia 中的 `IOBuffer` 类型提供了完善的缓冲机制，并能够以多种方式在内存中创建缓存。其内部的结构如表 12-3 所示。

表 12-3 IOBuffer 内部结构

字 段	类 型	意 义	字 段	类 型	意 义
data	Array{UInt8,1}	数据缓冲区	size	Int64	实际数据字节数
readable	Bool	可读标识	maxsize	Int64	最大容纳空间
writable	Bool	可写标识	ptr	Int64	游标位置
seekable	Bool	可回溯标识	mark	Int64	标签位置
append	Bool	可追加标识			

从表中可见，该类型内部以字节数组存储缓冲的数据，并具有可读、可写、可回溯及可追加的属性，同时在数据变更时会以 `size` 字段记录当前实际有效数据的字节数。虽然有这么丰富的结构，但是其中的很多字段会有默认值，所以在构造 `IOBuffer` 对象无须提





供所有的成员，主要的构造方法原型为：

```
IOBuffer([data::AbstractVector{UInt8}]; keywords... )  
IOBuffer(str::String)
```

其中，data 是某个已经存在的数据，作为参数输入时，便可以通过缓存区功能对其进行操作；如果不提供，建立的对象默认既可读也可写。键值参数 keywords 有如下几项：

- ❑ read、write、append：布尔型参数，用于约束缓存对象的读写权限；
- ❑ truncate：将缓存区清空为 0 长度；
- ❑ maxsize：限定缓冲区的大小，是缓冲区增长的上限；
- ❑ sizehint：建议的缓存区容量。

特别地，最后一个原型用于建立字符串 str 的只读缓存。

定义一个最大支持 3 字节的缓存，如下：

```
julia> b = IOBuffer(maxsize=3)  
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false,  
size=0, maxsize=3, ptr=1, mark=-1)
```

从字段取值可见该缓存可读可写，其内部的 data 内容此时为：

```
julia> b.data  
3-element Array{UInt8,1}:  
 0x00  
 0x00  
 0x00
```

尝试写入一些数据，例如：

```
julia> write(b, "abcdef") # 写入长度为6的字符串  
3 # 实际成功写入了3字节
```

此时，其中的 data 字段等内容为：

```
julia> b.data  
3-element Array{UInt8,1}:  
 0x61 # a  
 0x62 # b  
 0x63 # c  
  
julia> String(b.data) # 将缓冲区的字节序列转为字符串，便于查看  
"abc"  
  
julia> b.size # 容纳的有效数据字节数  
3  
  
julia> b.ptr  
4 # 游标值此时为b.size+1
```

可见，有效数据的字节数变成了 3 个，与 maxsize 字段的值一致。如果继续写入数据会发现：

```
julia> write(b, "def")  
0
```





可见, `write()` 函数成功写入的字节数为 0, 即表明没有成功写入过任何数据, 而且此时的 `data` 字段也没有更新。

实际上, 字段 `size` 不仅记录着有效数据的大小, 也同时与 `maxsize` 联合记录了可用缓存空间的大小。当 `size` 不小于 `maxsize` 时, 表明缓存区已满, 无法再写入更多的数据。缓存的作用是为了临时存取数据, 放入其中的内容应及时读取, 并将缓存空间释放以用于后续再次写入新数据。

若是采用通用的 `read()` 操作读取, 则无法达到这样的作用。该函数在获取缓存的内容时, 并不会对缓存对象进行任何的改变。为此, Julia 专门提供了 `take!()` 函数, 能在提取缓存内容的同时, 重置缓存到初始状态。例如:

```
julia> take!(b)
3-element Array{UInt8,1}:
 0x61
 0x62
 0x63
```

可见, 获得了其中的内容, 而且无须像 `read()` 那样需要显式地在操作前重置内部的读写游标。此时, 通过 `dump()` 查看 `b` 内部的状态 (已省略无关字段):

```
data: Array{UInt8}((3,)) UInt8[0x00, 0x00, 0x00]
size: Int64 0
ptr: Int64 1
```

可见, 缓存中的三个字节被读取, `data` 部分不再是原来的内容, 已经被随机值 (零值) 取代; 而 `size` 被重置为 0, 同时游标 `ptr` 也重置为 1。故此, 针对 `IOBuffer` 对象, 可以反复使用函数 `take!()` 不断地提取其中的内容。

不过只读的字符串缓存有些特殊性, 先看一个例子, 如下:

```
julia> s = "hello";
```

```
julia> buff = IOBuffer(s)
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false,
 size=5, maxsize=Inf, ptr=1, mark=-1)
```

该缓存区可读但不可写, 此时 `size` 为 5 且游标 `ptr` 位于初始位置。若通过 `take!()` 取走其中的内容之后, 再次查看其内部结构, 如下 (已省略无关内容):

```
julia> dump(buff)
data: Array{UInt8}((5,)) UInt8[0x68, 0x65, 0x6c, 0x6c, 0x6f]
size: Int64 5
ptr: Int64 1
```

能够发现 `size` 并不像之前那样被重置为 0, 而且 `data` 仍保留着被写入时的内容, 没有发生变化。如果我们反复地对其调用 `take!()` 函数, 也会发现内容并不会被替换, 每次都是一样的。之所以如此, 是因为 `String` 对象本身是不可变的, 而基于其建立的缓存对象也是只读的。





12.4 流的回溯

流通常存在于两个设备之间的传输过程中，例如，网络流是从一台机器传递数据到另外一台机器，而文件流则是数据向内存流动的过程。虽然流是字节的不断移动，但在实际应用中，总会有始有终，而且显而易见，流内的字节序列是有序的。

为了更好地对流进行访问或控制，流对象内部一般都会有游标结构，例如 `IOBuffer` 类中的 `ptr` 成员变量。这种结构的作用与传统硬盘中的磁头极为相似，用于标识当前操作处理到了流的哪个位置，例如读到了哪里、写到了哪里等。可以说，游标结构是对流进行控制、操作的重要基础。

Julia 中提供了丰富的以游标为基础的操作。我们可以使用函数 `eof()` 测试 IO 流是否结束，判断文件是否到达尾部；也可以使用 `seekstart()` 函数直接跳到起始位置，或者使用 `seekend()` 函数跳到流的结束位置；甚至能够通过 `seek()` 函数直接跳到某个位置，或使用 `skip()` 忽略头部若干位置直接移动到某处。在任何时候，我们都可以通过 `position()` 函数获得游标当前所处的位置。

在对流的处理过程中，尤其是读写操作，内部的游标值会自动移动。我们可以在其中的关键位置调用 `mark()` 进行标记（打标签），此后，如果要回到该处重新处理时，便可通过 `reset()` 将流的游标重置到该位置。如果在流的多个位置做了 `mark()` 调用，`reset()` 会回溯到最后一个标记位，同时会移除该标记信息。在此期间，我们可以使用 `ismarked()` 测试流是否有标记，或者使用 `unmark()` 移除标记。

需要注意的是，Julia 中流的读与写操作共享同一个游标，所以 `write()`、`read()` 等操作都会导致游标的移动。关于流中游标的各种操作总结在表 12-4 中。为了方便，在下面演示游标操作的过程中，使用 `IOBuffer` 对象进行示例。

表 12-4 流游标操作函数

操作函数	用途说明	操作函数	用途说明
<code>seekstart(io)</code>	移动到流的起始位置	<code>eof(io)</code>	判断流中是否有数据可读或是否在尾部
<code>seekend(io)</code>	移动到流的结束位置	<code>mark(io)</code>	在当前位置处进行标记
<code>seek(io, p)</code>	移动到绝对位置 <code>p</code>	<code>reset(io)</code>	重置游标到被标记处，无标记则抛出异常
<code>skip(io, offset)</code>	移动到当前位置的 <code>offset</code> 偏移处	<code>ismarked(io)</code>	判断流是否被标记过
<code>position(io)</code>	获取游标当前的位置	<code>unmark(io)</code>	移除流中的标记信息

现在，我们假设有一个只读的字符串缓存 `buff` 变量，其内容为“hello”，即：

```
julia> buff = IOBuffer("hello");
```

对其进行读取操作，如下：

```
julia> read(buff, 3)
3-element Array{UInt8,1}:
 0x68
 0x65
```



```
0x6c
```

此时，先检查 `buff` 游标所在的位置，并判断是否在尾部：

```
julia> position(buff)
3
```

```
julia> eof(buff)
false
```

```
julia> buff.ptr
4
```

可见，`position` 虽然为 3，但内部的游标 `ptr` 已经移动到了 4，以便后续操作能在其指向的字节处继续执行。

若在此时调用 `reset()` 则会报错，即：

```
julia> reset(buff)
ERROR: ArgumentError: Base.GenericIOBuffer{Array{UInt8,1}} not marked
```

因为之前我们并没有对其做过标记，也就是：

```
julia> ismarked(buff)
false
```

当前的 `position` 约处于中部位置，可以尝试做个标记以便下次能直接 `reset()` 到该处，即：

```
julia> mark(buff)
3
```

```
julia> ismarked(buff)
true
```

其中，`mark()` 调用时会返回被标记的 `position` 值。此时：

```
julia> buff.mark
3
```

可见内部结构中的 `mark` 值被设置为 3。

从此位置我们继续读取 `buff` 中的内容，即：

```
julia> read(buff,5)
2-element Array{UInt8,1}:
 0x6c
 0x6f
```

指定读 5 字节，但只返回 2 字节，说明可读数据只剩下了 2 个字节。查看此时的游标状态：

```
julia> position(buff)
5
```

```
julia> eof(buff)
true
```

```
julia> dump(buff)
# 已省略
ptr: Int64 6
```



```
mark: Int64 3
```

可见游标已移到了最后，ptr 已经大于 size 值。若再继续读取，则不会有更多内容，即：

```
julia> read(buff,5)
0-element Array{UInt8,1}
```

此前我们已经在中部做了标记，所以可以 reset() 到该位置，即：

```
julia> reset(buff)
3
```

```
julia> dump(buff)
```

```
# 已省略
```

```
ptr: Int64 4
```

```
mark: Int64 -1
```

可见，游标被重置到了 3 这个位置（对应的 ptr 值为 4），同时 mark 值被清空，被重置为 -1。此时若再读取，则：

```
julia> read(buff,5)
2-element Array{UInt8,1}:
 0x6c
 0x6f
```

当然，我们可以通过 seekstart() 函数直接将游标移动到起始位置，便可以再次读取到 buff 中的所有内容，即：

```
julia> seekstart(buff);
```

```
julia> read(buff,5)
5-element Array{UInt8,1}:
 0x68
 0x65
 0x6c
 0x6c
 0x6f
```

如果需要读取某个位置之后的指定内容，则可直接使用 seek() 函数，即：

```
julia> seek(buff, 3)
IOBuffer{data=UInt8[...], readable=true, writable=false, seekable=true, append=false,
  size=5, maxsize=Inf, ptr=4, mark=-1}
```

```
julia> read(buff,5)
2-element Array{UInt8,1}:
 0x6c
 0x6f
```

需要注意的是，起始位置 ptr=1 对应 seek() 中的位置参数 0（该函数是 0-based）。

另外一个 skip() 函数与 seek() 函数类似，不同的是，skip() 对游标的修改是基于当前游标位置的（相对移动），而不是像 seek() 那样直接移动到绝对位置。例如：

```
julia> seek(buff, 3)
IOBuffer{data=UInt8[...], ..., ptr=4, mark=-1}

julia> skip(buff, 1)
```



```
IOBuffer(data=UInt8[...], ..., ptr=5, mark=-1)
```

```
julia> read(buff, 5)
1-element Array{UInt8,1}:
 0x6f
```

其中, `seek()` 将游标置于绝对位置 3, 再由 `skip()` 以该位置为基础向后相对移动 1 个位置, 之后便只能读取剩余的一个字节内容了。

基于上述对游标各种操作的了解, 下面提供一段程序完整地演示游标的各种操作:

```
julia> buff = IOBuffer("abcdefghi")
IOBuffer(data=UInt8[...], readable=true, writable=false, seekable=true, append=false,
  size=9, maxsize=Inf, ptr=1, mark=-1)

julia> seek(buff, 0);

julia> while !eof(buff)
    ch = read(buff, Char);
    println("position ", position(buff), " is ", ch, ", isend:", eof(buff));
    if ch == 'c'
        mark(buff)
        skip(buff,1)
    end
    if position(buff) >= 6 && ismarked(buff)
        reset(buff)
    end
end
position 1 is a, isend:false
position 2 is b, isend:false
position 3 is c, isend:false
position 5 is e, isend:false
position 6 is f, isend:false
position 4 is d, isend:false
position 5 is e, isend:false
position 6 is f, isend:false
position 7 is g, isend:false
position 8 is h, isend:false
position 9 is i, isend:true
```

可见, 由于 `mark()` 函数、`skip()` 与 `reset()` 函数的参与, 输出的结果出现了多次跳跃。

12.5 序列化

在程序处理中往往需要定义各种类型以便进行不同层次的数据转换, 而在此之前通常需要从流 (例如文件) 中加载数据到预先定义的结构中, 转化完成后又通常需要输出到流中, 即传输出去。这就涉及常用类型与字节流之间的变换。

在模块 `Serialization` 中, Julia 提供了 `serialize()` 和 `deserialize()` 分别用于类型到流的转换或流到类型的转换。为了方便说明, 使用缓存类型 `IOBuffer` 模拟某个输入输出流, 并假设待处理的数据为字符串, 内容为 “hello world”, 即:


```
julia> s = "hello world";          # 字符数为11
```

为了能够将其转换到流中，先定义一个可写的缓存类型：

```
julia> sbuff = IOBuffer()
IOBuffer(data=UInt8[...], readable=true, writable=true, seekable=true, append=false,
size=0, maxsize=Inf, ptr=1, mark=-1)
```

从反馈的信息可见，sbuff 的 readable 与 writable 值均为 true，所以既可读也可写。此时其缓存内容为：

```
julia> sbuff.data
32-element Array{UInt8,1}:          # 默认先分配32字节
 0x00
 0x00
 # 已省略
```

然后对 sbuff 以及字符串 s 调用 serialize() 函数：

```
julia> using Serialization
```

```
julia> serialize(sbuff, s)
```

此时利用 dump() 查看 sbuff 的内部结构，如下：

```
julia> dump(sbuff)
Base.GenericIOBuffer{Array{UInt8,1}}
 data: Array{UInt8}{(32,)} UInt8[0x37, 0x4a, 0x4c, 0x07, 0x04, 0x00, 0x00, 0x00,
 0x21, 0x0b ... 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00]
 readable: Bool true
 writable: Bool true
 seekable: Bool true
 append: Bool false
 size: Int64 21
 ptr: Int64 22
 # 其他已省略
```

```
julia> sbuff.data
32-element Array{UInt8,1}:
 0x37          # 描述头开始位置
 0x4a
 0x4c
 0x07
 0x04
 0x00
 0x00
 0x00
 0x21
 0x0b
 0x68          # 数据区开始位置
 0x65
 0x6c
 0x6c
 0x6f
 :
 0x72
 0x6c
```

```

0x64
0x00
0x00
0x00
0x00
... # 其他已省略

```

可见，size 大小比原始字符串的字节数多 10 个，因为 Julia 自动增加了描述头信息，包括标识符号、协议版本、字节序、平台位数等，有兴趣的读者可以参考相关资料，此处不做赘述。

此后，我们便可以基于 sbuff 进行后续的处理，例如传输、存储等操作。当然，在获得这样的字节流后，便对其实施反操作，即反序列化，“解译”出原类型数据，即：

```

julia> deserialize(sbuff)
"hello world"

julia> typeof(ans)
String

```

可见，原字符串被恢复了。

除了字符串，序列化与反序列化操作广泛地支持 Julia 中的各种类型。例如，某个复合类型定义如下：

```

julia> struct Values
    a::String
    b::Int32
    c::Float64
    d::Array{Float32}
end

```

```

julia> v = Values("hello", 12, 3.5, [1 2 3 4])
Values("hello", 12, 3.5, Float32[1.0 2.0 3.0 4.0])

```

我们需要将其输入/输出到如下的流对象：

```

julia> stream = IOBuffer();

```

首先对其进行序列化，如下：

```

julia> serialize(stream, v)

```

查看下 stream 中的内容，如下：

```

julia> dump(stream)
Base.GenericIOBuffer{Array{UInt8,1}}
data: Array{UInt8}((67,)) UInt8[0x37, 0x4a, 0x4c, 0x07, 0x04, 0x00, 0x00, 0x00,
    0x34, 0x10 ... 0x00, 0x40, 0x00, 0x00, 0x40, 0x40, 0x00, 0x00, 0x80, 0x40]
size: Int64 67
ptr: Int64 68
# 其他已省略

```

可见，对象 v 序列化后，占据了 67 个字节（包括头信息）。

假设经过远程传输后，该对象的字节流被远端的另外一个同名流对象接收到，此时便可对其中的内容进行反序列化，即：

```
julia> deserialize(stream)
Values("hello", 12, 3.5, Float32[1.0 2.0 3.0 4.0])
```

可见, Values 类型的对象 *v* 被原样恢复。

12.6 网络通信

关于服务端与客户端的操作,可以简单地描述为:服务端程序监听某个端口等待客户端的连接,如果条件符合便接受该次连接;客户端只需提供服务端的地址与端口,实现连接行为即可。在两者建立通信后,便可基于各自获得的套接字对象,以流操作的方式进行数据的读写。

在 Julia 语言中,网络通信的功能封装在 Sockets 模块中。使用 Julia 语言进行网络编程极为简单,基本不会涉及太多底层的内容。

在服务端如果要建立监听任务,执行以下命令即可:

```
listen([addr, ]port::Integer; backlog::Integer=BACKLOG_DEFAULT) -> TCPServer
```

其中, *addr* 指定本地网卡的 IP 地址,默认为 localhost,若值为 IPv4(0) 或 IPv6(0) 则可监听所有的网卡; *port* 指定监听的端口号;参数 *backlog* 用于限定服务端拒绝更多连接前允许挂起等待的连接数,默认为 511 个。

例如,执行下面的命令便可监听默认网卡 localhost 的 2000 端口:

```
julia> using Sockets

julia> server = listen(2000)
Sockets.TCPServer(Base.Libc.WindowsRawSocket(0x0000000000000384) active)
```

其中 Sockets.TCPServer 类型的继承树为:

```
Sockets.TCPServer <: Base.LibuvServer <: Base.IOStream
```

可见,其也是基于 Libuv 库的。

除了能以上例的方式建立监听任务外,还可以下面的方法实现:

```
julia> listen(2000) # 监听localhost:2000, IPv4协议
Sockets.TCPServer(Base.Libc.WindowsRawSocket(0x0000000000000374) active)
```

```
julia> listen(ip"127.0.0.1", 2000) # 等效于上一个用法
Sockets.TCPServer(Base.Libc.WindowsRawSocket(0x00000000000003f0) active)
```

```
julia> listen(ip ":::1", 2001) # 监听 localhost:2001, IPv6协议
Sockets.TCPServer(Base.Libc.WindowsRawSocket(0x0000000000000598) active)
```

```
julia> listen(IPv4(0), 2002) # 监听所有IPv4网络接口上的2002端口
Sockets.TCPServer(Base.Libc.WindowsRawSocket(0x0000000000000644) active)
```

```
julia> listen(IPv6(0), 2003) # 监听所有IPv6网络接口上的2003端口
Sockets.TCPServer(Base.Libc.WindowsRawSocket(0x00000000000006d4) active)
```

```
julia> listen("testsocket") # 监听UNIX域套接字或Windows命名管道
```



其中，IP 地址字符串的前缀“ip”是专门标识符，用于表达 ip 地址字符串。

客户端此时便可以通过调用 `connect()` 函数连接到服务端，该函数的原型为：

```
connect([host], port::Integer) -> TCPSocket
```

其中，`host` 为服务端的地址；`port` 为端口号，需与服务端监听的端口号一致。例如：

```
julia> sockc = connect(2000)
TCPSocket(Base.Libc.WindowsRawSocket(0x0000000000000304) open, 0 bytes waiting)
```

连接到 `localhost` 的 2000 端口，成功后返回 `TCPSocket` 对象，即套接字，其类型继承树为：

```
TCPSocket <: Base.LibuvStream <: IO
```

可见，其与标准流具有相同的父类型 `Base.LibuvStream`，所以也是流的一种。

当然连接建立函数还有其他的形式，例如，可以指定域名：

```
julia> connect("www.baidu.com", 80)
TCPSocket(Base.Libc.WindowsRawSocket(0x00000000000007c8) open, 0 bytes waiting)
```

若是需要获得某个域名的 IP 地址，可以通过 `getaddrinfo()` 函数实现，例如：

```
julia> getaddrinfo("www.baidu.com")
ip"115.239.210.27"
```

当某个客户端 `connect()` 成功后，服务端便可通过 `accept()` 函数接受该次连接，即：

```
julia> socks = accept(server)
TCPSocket(Base.Libc.WindowsRawSocket(0x000000000000037c) open, 0 bytes waiting)
```

可见，此时服务端同样创建了 `TCPSocket` 类型的套接字对象，同样是一种流类型。

至此，服务端与客户端之间成功建立了连接，而两者通信的依据便是各自的套接字流对象，可以通过该对象进行读写，实现数据的传输。

假设服务端执行 `readline(socks)`，便会阻塞等待连接的客户端写入数据。如果客户端往套接字对象写入内容，如下：

```
julia> write(sockc, "client's message\n")
17
```

之后，服务端便会接收到该字符串，并在发现换行符 `\n` 后停止阻塞，返回获得的行字符串数据，即：

```
julia> readline(socks)
"client's message"
```

网络数据传输同样支持复杂类型。为了能够实现类型的相互解译和转换，数据写入时需使用 `serialize()` 函数，相应地需使用 `deserialize()` 函数进行读取操作。

例如，要将上文中复合类型 `Values` 的一个对象从客户端传输到服务端，我们先在服务端执行 `deserialize(socks)` 语句，阻塞等待数据传入，然后在客户端写入该类型的数据：

```
julia> using Serialization
```




```
julia> v = Values("hello", 12, 3.5, [1 2 3 4]);  
  
julia> serialize(sockc, v)
```

此时服务端会接收到数据，并停止阻塞。

但如果出现以下信息：

```
julia> deserialize(socks)  
ERROR: UndefVarError: Values not defined
```

说明服务端虽然成功接收到客户端数据，但类型未定义。所以该复合类型的定义需同时在服务端代码中存在。在服务端对该类型声明后，便可成功解析接收到的数据，即：

```
julia> deserialize(socks)  
Values("hello", 12, 3.5, Float32[1.0 2.0 3.0 4.0])
```

上文我们一步步地介绍了 Julia 网络编程的基本过程。下面可以给出一个比较全面的简单服务端代码：

```
julia> using Sockets, Serialization  
  
julia> begin  
    server = listen(2000)  
    while true  
        socks = accept(server)  
        println("One client reached!\n")  
        while isopen(socks)  
            data = deserialize(socks)  
            println("Read: ", data)  
            serialize(socks, data)  
        end  
    end  
end
```

该程序监听 2000 端口，当接收到客户端连接后，会打印 “One client reached!” 信息。之后便会读取并反序列化接收到的数据，打印该数据后再将该数据序列化到套接字对象中，反馈到客户端。

当客户端执行：

```
julia> using Sockets, Serialization  
  
julia> sockc = connect(2000)  
TCPSocket(Base.Libc.WindowsRawSocket(0x0000000000000344) open, 0 bytes waiting)
```

服务端会打印以下信息：

```
One client reached!
```

客户端再执行：

```
julia> serialize(sockc, "client's msg 1")
```

服务端会输出以下结果：

```
Read: client's msg 1
```



此时客户端执行反序列化，接收到服务端的反馈信息，如下所示：

```
julia> deserialize(sockc)
"client's msg 1"
```

如果客户端持续执行：

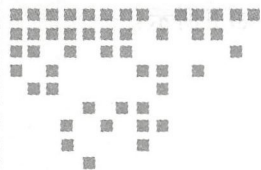
```
julia> serialize(sockc, "client's msg 2")
```

```
julia> serialize(sockc, "client's msg 3")
```

则服务端会相应地输出结果，如下所示：

```
Read: client's msg 2
Read: client's msg 3
```

至此，示例代码的运行过程便演示完成了。如果结合后文介绍的并行机制，我们能够实现更为灵活和复杂的网络程序。



组织结构

13.1 模块

模块是 Julia 特有的组织结构，不同的模块构成了相互隔离的变量工作区，所以创建一个模块时同时会引入一个新的全局域（Global Scope），成为一个独立的名字空间。此时在新的模块内，能够创建顶层级别的声明或定义（即全局性变量），不必担心会与其他模块中的名字冲突。

13.1.1 基本定义

定义一个模块的基本语法为：

```
module 模块名
    # 模块内容
end
```

另外，可使用 `baremodule` 关键字定义模块，后面再介绍。

类似于命名空间，模块中除了变量外，还可以声明或定义常量、类型、函数等，从而实现功能结构的组织划分，便于维护和发布。而且，在模块中可以引入其他外部模块，从而能使用其中已经定义好的对象，也可以显式地导出对象，提供给外部使用。

下面的例子说明了模块的用法：

```
module MyModule

using Lib
using BigLib: thing1, thing2
```



```
import Base.show

export MyType, foo           # 导出以供外部使用

struct MyType                # 定义的类型
    x
end

bar(x) = 2x                  # 定义的函数

foo(a::MyType) = bar(a.x) + 1 # 定义的函数

show(io::IO, a::MyType) = print(io, "MyType $(a.x)")

end
```

其中，创建了名为 `MyModule` 的模块，内部定义了复合类型 `MyType` 及两个函数 `bar()`、`foo()`。在模块中定义各种内容的过程中，Julia 在句法上并没有对齐或者缩进的特殊要求。

下面以 `MyModule` 为例，具体说明其中一些关键字的意义：

- ❑ 关键字 `export`。该关键字之后以逗号隔开的名字会被导出，可以被其他模块引入以便使用。未列于其中的对象则是模块私有的，不能被外部自动发现，例如 `MyModule` 中的 `bar` 函数。
- ❑ 关键字 `using`。该关键字用于提供外部模块及其成员以扩充名字解析的范围。如果未能在当前模块找到某个名字的声明，则会在 `using` 之后提供的名单中搜寻，如果发现，便会引入进来。
- ❑ 关键字 `import`。使用的语法与 `using` 相同，但不会将模块加入到名字解析的搜寻范围内，而且一次只能引入一个名字。另外，若要为已有的函数扩展新方法，只能使用 `import` 引入该函数的名字。例如 `MyModule` 中的 `Base.show()` 函数。

下面再以一个例子具体说明各种引入方式的区别，如下：

```
module Md1

export x, y

x() = "x"
y() = "y"
p() = "p"

end
```

其中，模块 `Md1` 定义了三个函数，但只导出了其中的两个。在使用不同的方式引入该模块时，会有不同的效果，具体如表 13-1 所示。



表 13-1 模块引入方式对比

引入命令	引入到名字空间的内容	函数扩展适用性
using Mdl	所有导出的名字 (x 及 y), Mdl.x, Mdl.y 及 Mdl.p	Mdl.x, Mdl.y 及 Mdl.p
using Mdl.x, Mdl.p	x 及 p	
using Mdl: x, p	x 及 p	
import Mdl	Mdl.x, Mdl.y 及 Mdl.p	Mdl.x, Mdl.y 及 Mdl.p
import Mdl.x, Mdl.p	x 及 p	x 及 p
import Mdl: x, p	x 及 p	x 及 p

还有一点需要说明：无论用哪种方式引入了其他模块的名字，该名字便出现在当前模块的名字空间内，所以不能再重复使用。

在引入名字的过程中，有两类名字比较特别：

一种是只有符号的名字，例如运算符 `+`，不能使用 `Base.+` 的方式，而要采用 `Base.:+` 这种 `Symbol` 的方式表达；如果运算符不仅有一个字符，则需要使用圆括号进行限定，例如 `==` 需要使用 `Base.:(==)` 表达。

另外一种宏，名字中有 `@` 标识符，在引入时也需带着标识符，例如 `import Mod.@mac`，调用其他模块的宏时，也需如此，即 `Mod.@mac` 或 `@Mod.mac`。

13.1.2 标准模块

Julia 内置了三个标准模块：`Core`、`Base` 和 `Main`。其中的 `Core` 模块包括了 Julia 语言内置的所有标识符、运算符、类型及其最基本的操作函数，是 Julia 的核心部分，所以每个定义的模块都会隐式地引入该模块。`Base` 模块是 Julia 的标准库，因为大部分情况都被使用，所以新建的模块也会隐式地引入该模块。`Main` 模块仅是一个默认的最高层命名空间，内部除了自动引入 `Core` 与 `Base` 模块外，初始时没有其他定义或声明的内容；Julia 的 REPL 便处于该模块中。

对于已经定义的模块，可以使用 `varinfo()` 函数查看其中的名字列表，不仅包括其中声明定义的种种变量，也会列出引入的模块等信息。例如：

```
julia> varinfo(Core)
name                               size summary
-----
<:                                0 bytes typeof(<:)
===                               0 bytes typeof(===)
AbstractArray                      80 bytes UnionAll
AbstractChar                      172 bytes DataType
AbstractFloat                     172 bytes DataType
AbstractString                    172 bytes DataType
Any                               172 bytes DataType
ArgumentError                    188 bytes DataType
Array                             80 bytes UnionAll
AssertionError                   188 bytes DataType
```



```
Bool          172 bytes DataType
BoundsError   196 bytes DataType
Char          172 bytes DataType
Core          Module
Cvoid         172 bytes DataType
DataType      340 bytes DataType
# ... 已省略
nothing       0 bytes Nothing
setfield!     0 bytes typeof(setfield!)
throw         0 bytes typeof(throw)
tuple         0 bytes typeof(tuple)
typeassert    0 bytes typeof(typeassert)
typeof        0 bytes typeof(typeof)
undef         0 bytes UndefinedInitializer
```

如果该函数不提供参数，则返回当前默认模块（一般为 Main）的内容，例如：

```
julia> varinfo()
name          size summary
-----
Base          Module
Core          Module
InteractiveUtils 157.063 KiB Module
Main          Module

julia> a = 10;

julia> b = x->x*2
(::#3) (generic function with 1 method)

julia> varinfo()
name          size summary
-----
Base          Module
Core          Module
InteractiveUtils 157.063 KiB Module
Main          Module
a             8 bytes Int64
ans           0 bytes getfield(Main, Symbol("##3#4"))
b             0 bytes getfield(Main, Symbol("##3#4"))
```

其中列出了 Main 中的所有名字，在定义了两个变量（a 与 b）之后，便也加入在该模块的名字列表中。

如果在定义模块的时候，不希望隐式地引入 Base 等，可以采用 `baremodule` 定义新的模块，例如：

```
baremodule Mod
end
```

此时，如果查看其中的定义会发现：

```
julia> varinfo(Mod)
name          size summary
-----
Mod           16 bytes Module
```



其中没有自动引入 Core 和 Base 模块。如果要使用这两个模块中的内容，便需要显式地使用引入语句。

13.1.3 模块路径

当使用 `using Foo` 这样的语句引入某个模块时，系统会先在 Main 模块相关的所有代码中搜索。如果未找到，系统会再执行 `require("Foo")` 命令，尝试在已安装的包中进行加载。

然而，一些模块会包含子模块，即模块嵌套了另外的模块，而这些子模块对 Main 来说不是直接可见的，如果要访问或引入可采用两种方式：一种是使用绝对路径，即在引入时罗列出模块的整个嵌套路径，例如 `using Base.Sort`；第二种便是使用相对路径，例如：

```
module Parent

    module Second

        module Utils

            end

            using. Utils      # Utils之前有个点

        end

        # 其他定义声明

    end
```

其中，Parent 内部模块包括一个名为 Second 的子模块，在 Second 内部引用同层次的 Utils 模块时，只需在 `using` 之后模块名 Utils 之前附加一个英文句号（当前路径）。若是使用更多的句号，便可继续上溯模块的层次，例如 `using ..Parent` 会在 Second 模块的上级中寻找 Parent 模块。



注意 这种相对路径的方式只适用于 `using` 及 `import` 语句。

13.1.4 预编译

我们经常发现，加载大的模块常需要几秒，这是因为在执行前需要编译大量的代码。为此，Julia 支持对模块进行预编译，以减少这种时间。默认情况下，Julia 会在模块被首次加载时进行自动的预编译，这相当于在模块所处文件的顶部增加了一个特别的语句，即 `__precompile__()` 函数。而生成的预编译版本模块会被存储于 `Base.LOAD_CACHE_PATH` 指定的目录下。除此之外，也可在 REPL 中自行调用 `Base.compilecache(modulename)` 命令执行这个预编译过程。

之后，如果发生模块变更、包含文件变更、`include_dependency(path)` 显式引入



了其他依赖等情况，或 Julia 程序被构建时，预编译的模块会再次被重新编译。

13.2 模块与脚本文件

Julia 的各种表达式语句可以写在脚本中统一执行。脚本文件一般以 `.jl` 为扩展名，文件名则可以任起。

若要执行某个脚本中的命令，只需在控制台或 Shell 中执行：

```
julia script.jl
```

或者在 REPL 中执行：

```
julia> include ("script.jl")
```

其中，`script.jl` 是创建的某个脚本文件名称（可以用绝对路径或相对路径给出）。执行命令时，被调用脚本文件中的语句会按序逐一执行。

1. 脚本包含

出于结构划分等需要，可以将程序分散在不同的文件中，然后在使用中，将需要的各种脚本通过 `include()` 函数包含在运行脚本中。该函数可在参数中提供所需脚本的绝对路径或相对路径。

例如，有三个脚本分别为 `parta.jl` 文件、`partb.jl` 文件及 `part1.jl` 文件，各自的内容如下：

```
##### parta.jl
const a = Int32(2)
```

及：

```
##### part1.jl
A = rand(3,3)
```

还有：

```
##### partb.jl
include("part1.jl")

function display(array::Array)
    @show array
end
```

而脚本 `script.jl` 中的内容为：

```
##### script.jl
include("parta.jl")
include("partb.jl")

display(A)

B = A.^a

display(B)
```




其中, `partb.jl` 包含了 `part1.jl` 脚本; `script.jl` 包含了 `parta.jl` 与 `partb.jl` 脚本。

通过 `julia script.jl` 执行名为 `script.jl` 这个主脚本后, 会得到以下的结果:

```
array = [1 2; 3 4; 5 6]
array = [1 4; 9 16; 25 36]
```

由此可见, Julia 脚本文件可以通过 `include` 建立各种组织结构关系, 因此经常用来进行包代码文件的管理。虽然 Julia 中没有“主函数”这种结构, 但却有“主脚本”的用法, 例如上例中的被调用执行的 `script.jl` 便成为了主脚本, 它是运行过程的启动点。

2. 与模块的关系

虽然脚本文件是功能分割的方式之一, 与模块有些相似, 但文件与模块并没有直接的关联, 相互之间也没有约束关系——一个模块可以分散到多个文件中, 或一个文件中可有多个模块。

例如, 在一个模块中 `include` 多个文件, 如下:

```
module Foo

include("file1.jl")
include("file2.jl")

end
```

实际是将一个模块的部分分散在不同的文件中。或是在一个文件中定义了两个模块:

```
module Normal
    include("mycode.jl")
end

module Testing
    include("safe_operators.jl")
    include("mycode.jl")
end
```

这种方式中, 不同的模块均 `include` 了名为 `mycode.jl` 文件, 故两个模块会运行同一份代码。利用这种方式, 可以对正式环境与测试环境实现“糅合”或分离, 从而能够进行高效地开发。

13.3 变量域

在编程语言中, 变量域 (Variable Scope) 概念是必然要遇到的常规问题。在一个代码块中, 变量是否可见、是否可用, 都和域控制有关。而变量定义在不同的位置, 就有着不同的有效域, 也影响着在某个代码块中是否能使用它。

另外, 对变量域的限制也是名字机制的一部分, 可以避免命名重复。在隔离的不同名字空间中, 变量名、函数名、类型名等可以是相同的, 甚至可以有相同的原型声明。同名及同原型是否指代同一个实现或对象, 由域控制规则确定。

在变量域方面, Julia 采用的是语法域 (Lexical Scoping) 规则, 是一种与 Bash 的动态

域 (Dynamic Scoping) 不同的静态域。在该规则中, 域的传播和影响由代码文本所处的位置决定。为了后续说明的方便, 先从不同的角度区分两个概念: ①作用域, 某个名称能被使用的范围, 一般是指该名称首次出现之后或之下的代码域; ②可见域, 某个代码处可见的名称集合或可访问的范围, 一般指该行代码之上或之前的各种语句, 不过由于代码块或控制结构的存在, 这个范围有“孔洞”, 并不是全覆盖的。

关于语法域规则的特点, 我们以函数为例——函数的可见域并不会继承调用者的可见域, 而是来自于其定义处。例如, 一个函数 `foo()`, 使用了 `x` 变量:

```
julia> module Bar
    x = 1      # 定义处的x变量
    foo() = x
end;
```

定义完成后, 进行调用:

```
julia> import .Bar

julia> x = -1;      # 调用处的x变量

julia> Bar.foo()
1
```

可见, `foo()` 内部所使用的变量 `x` 来自于其定义处的模块内部, 而不是调用处的那个 `x` 变量。

Julia 中的域大致分为两类——全局域 (Global Scope) 和局部域 (Local Scope), 而后者又可分为可嵌套 (Nesting) 及不可嵌套两种, 详细见表 13-2。

表 13-2 作用域与代码块关系

作用域分类	对应的代码块或结构
全局域	module, baremodule, REPL (Main 模块)
局部域 (可嵌套)	for, while, 推导式, try-catch-finally, let, 函数 (包括匿名函数、do 代码块)
局部域 (不可嵌套)	struct (无论是否 mutable 类型), macro

所谓嵌套, 指的是内部又存在一个代码结构, 例如多层循环便是循环的嵌套, 但复合类型定义的内部不可再嵌套其他结构。

从表中可以发现 `begin-end` 和 `if-end` 两个结构并未出现, 因为它们不会引入新的域。以 `begin` 代码块为例, 如下:

```
julia> begin
    a = 1;
end
```

```
julia> a
1
```

```
julia> b = 10;
```

```
julia> begin
    b = 3;
```

```
end

julia> b
3
```

其中，内外变量 `a` 与 `b` 并没有因为 `begin` 结构的存在而在操作上有什么限定。可见，`begin` 结构的存在并没有对内外变量的控制域产生任何影响。

关于 Julia 中各种域的特性，下文会详细说明。

13.3.1 全局域

全局域可认为是最外延、最大的域，也是完全隔离、相互独立的。在目前的 Julia 中，模块是创建全局域的唯一方式。当然，模块可以通过 `import` 或 `using` 等操作符，将其他模块的作用域引入到自己的可见域中。

下面通过示例说明模块的作用域情况：


```
julia> module A
    a = 1          # 模块A作用域下的全局性变量a
end;

julia> module B
    module C
        c = 2
    end

    b = C.c        # 通过点操作符访问一个嵌套域内的名字
    import ..A      # 使得A进入可见域
    d = A.a
end;

julia> module D
    b = a          # 报错：D的全局域与A的全局域是隔离的
end;
ERROR: UndefVarError: a not defined

julia> module E
    import ..A      # 使得A进入可见域
    A.a = 2         # 抛出如下异常：不能改变其他全局域内的变量
end;
ERROR: cannot assign variables in other modules
```

 **注意** 变量的绑定关系只能在所处的全局域内被改变，而不能通过外部的模块进行。

13.3.2 局部域

大多数的代码块或结构会引入一个新的局部域，如表 13-2 所示。通常来说，局部域会从父级可见域（结构外部，包括全局域）继承其中的所有名称，并能够进行读写访问。

对于局部域而言，由于并不是独立的命名空间，所以其内部的名称不会反向传播到父级作用域，即，如果某个名称在局部域内是首次出现的且未在父级域出现过，则该名称对

父级域不可见，即便同名也是不同的对象。

以局部域 `for` 循环为例（为叙述方便，示例均假设全局域是干净的，即所涉及的名称从未出现过）：

```
julia> for i = 1:2
    z = i
end

julia> z
ERROR: UndefVarError: z not defined
```

其中，变量 `z` 是 `for-end` 结构体中新引入的变量，所以在 `for` 结构体的外部不可访问，即 `z` 并没有进入 `for` 的父级作用域中。而且，即使 `for` 结构体的名称已经在外部域内出现过，该名称也会作为局部变量处理，不会修改外部同名变量绑定的内容，例如：

```
julia> z = 1;                # 变量z首次出现，并绑定到值1

julia> for i = 1:2
    z = i                    # 修改z的值，最后一次循环z的值应为2
end

julia> z                      # for循环外部访问z，值并未改变
1
```

但如果需要在 `for` 内部使用该 `z` 的内容，并能够控制或修改其内容，则可利用 `global` 关键字标识该变量。例如：

```
julia> z = 1;                # 变量z首次出现，并绑定到值1

julia> for i = 1:2
    global z                 # 声明使用的是全局变量z
    z = i
end

julia> z                      # 外部的变量z被改变
2
```

当然，在这个场景中，`global` 使用的位置并没有过多的约束，下述的方式也是可以的：

```
julia> z = 1;

julia> for i = 1:2
    z = i
    global z
end

julia> z
2
```

相对地，还有一个 `local` 关键字，可以限定某个变量只能在当前域可见，不会受到外部（父级或全局域）名称的干扰，例如：

```
julia> local z = 0;

julia> for i = 1:2
```



```
println(z)
end
ERROR: UndefVarError: z not defined
```

从中可见，虽然 z 在全局域 REPL 中，但对 `for` 结构内部却是不可见的。

对于局部域而言，总会从外部域（父级或全局域）继承所有变量，除非该变量被显式地表示了 `local` 关键字，并能够读取这些变量的值。但如果在局部域内出现对继承的变量进行赋值操作时，便会创建新的变量而不会修改原来的同名变量。以函数为例，再次说明此点：

```
julia> x, y = 1, 2; # 全局域的两个变量

julia> function foo()
    x = 2 # 因为赋值操作，导致x本质上已经不同于前面语句中的全局变量，
           # 而是新的同名局部变量
    return x + y # 此时y因未被赋值操作干扰，所以仍对应着全局域的那个y
end;
```

```
julia> foo()
4
```

```
julia> x
1
```

所以，如果要在局部域内修改外部域中的变量，需显式地标识为 `global`，例如：

```
julia> x = 1; # 全局变量

julia> function foobar()
    global x = 2 # 对全局变量进行修改
end;
```

```
julia> foobar();
```

```
julia> x # foobar内部的修改操作对外部可见
2
```

不过，对类似于嵌套函数定义这种情况则会有所不同，例如：

```
julia> x, y = 1, 2; # 全局变量x和y

julia> function baz() # 上层函数
    x = 2 # 赋值导致生成新的同名局部变量
    function bar() # 内层嵌套的函数
        x = 10 # 修改上层函数的局部变量
        return x + y # y只读，取全局变量值；等效于 10+2 = 12
    end
    return bar() + x # x=10已经被bar()修改，等效于12+10
end;
```

```
julia> baz()
22
```

```
julia> x, y # 作为全局变量，始终未被改变
(1, 2)
```

可见，内层函数的修改能够直接传播到外层函数中，这种差异在开发中是需要注意的。

在开发中，作用域或可见域是非常重要的概念，Julia 的各种代码块在这方面又有一些特别的地方，所以需要清楚每个代码块的特点，才能准确地开发出预期的功能。

13.3.3 let 关键字

Julia 中的 `let` 关键字是一种特殊的代码块，能够接受以逗号分割的赋值语句或变量名作为参数，并将其中涉及的变量重新分配内存，同时转为局部性的变量，使代码块中的复合表达式对其所做的操作只在内部可见。用法如下例所示：

```
julia> x, y, z = -1, -1, -1;

julia> let x = 1, z
    println("x: $x, y: $y") # 同名x被新建为局部变量，而y未在let声明，故仍保持为全局性变量
    println("z: $z")        # 同名z被新建为局部变量，但未赋值所以报undefined错误
end
x: 1, y: -1
ERROR: UndefVarError: z not defined
```

可见，`let` 的参数相当于使用了 `local` 关键字，但必须初始化才可正常使用。从本质上，这是因为 `let` 结构能够创建独立的局部域。看下面的例子：

```
julia> let
    x = 1
end
1

julia> x
ERROR: UndefVarError: x not defined
```

其中，`x` 仅在 `let` 中定义，而在外部域是不可见的，所以 `let` 的实现部分构成了局部的隔离空间，其内部出现的名称都无法被外部直接使用。在 `let` 嵌套时，可以结合 `local` 关键字，达到作用域隔离的目的，例如：

```
julia> let
    local x = 1
    let
        local x = 2
    end
    x
end
1
```

其中，内层 `let` 对 `x` 的修改并没有影响到外层 `let` 中的 `x` 变量，所以内外两个 `x` 虽然同名，但实际是对应不同内存区的变量。需要注意的是，在这种嵌套的 `let` 结构中，若同时使用 `global` 关键字与 `local` 时，不恰当的用法会导致错误发生，例如：

```
julia> let
    local x = 1
    let
        global x = 2
    end
end
```

```

x
end
ERROR: syntax: `global x`: x is local variable in the enclosing scope

```

其中, 变量 `x` 先被标识为 `local` 又在内层 `let` 中被标识为 `global`, 导致了语法错误。

至此, 我们对 `let` 的特性进行总结:

- ❑ `let` 内部的语句为复合表达式, 可以返回最后一个子表达式的值;
- ❑ `let` 会引入新的局部域, 内部新建的变量对外部不可见, 除非以 `global` 标识;
- ❑ `let` 后定义的变量为局部变量, 会切断外部同名变量的关系。

此外另需说明的一个特点是, `let` 后的赋值表达式是依序求值的: 在左操作数作为新局部变量引入前, 会先对右操作数(表达式)求值并作为它的初始值。例如:

```

julia> x,y,z = -1,-1,-1;

julia> let x=1, y=x+1, z=y+2          # 后面的赋值表达式能够使用前面表达式的变量值
    println("x: $x")
    println("y: $y")
    println("z: $z")
end
x: 1
y: 2
z: 4

```

所以 `let x=x` 是有效的, 因为两个 `x` 意义并不相同, 有着不同的内存地址, 不过需右侧的 `x` 在父级可见域存在。

下面举例介绍 `let` 的用途, 如下:

```

julia> Fs = Vector{Any}(undef, 2); i = 1; # 创建2元的函数的数组, 及一个变量i

julia> while i <= 2                    # 使用全局变量i作为判断条件
    Fs[i] = ()->i                      # 创建匿名函数并赋值给数组的第i个元素
    global i += 1                     # 试图使用i进行索引, 并递增
end

julia> Fs[1]()
3

julia> Fs[2]()
3

```

上述代码试图定义一个函数, 直接返回函数在数组中的索引值, 但结果却不是预想的那样。为了找到失败原因, 查看 `Fs` 的内部结构:

```

julia> dump(Fs)
Array{Any}((2,))
 1: getfield(Main, Symbol("##3#4"))() (function of type getfield(Main, Symbol("##3#4")))
 2: getfield(Main, Symbol("##3#4"))() (function of type getfield(Main, Symbol("##3#4")))

julia> Fs[1] == Fs[2]
true

```

尝试在外部改变 `i` 的值, 再次调用函数:

```
julia> i += 2
```

```
5
```

```
julia> Fs[1]()
```

```
5
```

```
julia> Fs[2]()
```

```
5
```

由此可以得出结论, 在定义函数 $Fs[i] = () \rightarrow i$ 时, i 变量名被绑定到了函数的定义中, 而不会记录 i 的值的变化的, 即不会受到 `while` 内部表达式 $i+=1$ 的影响。

为了能够得到如下的效果:

```
julia> Fs[1]()
```

```
1
```

```
julia> Fs[2]()
```

```
2
```

我们利用 `let` 语法的特性, 改造上述的定义。

```
julia> Fs = Vector{Any}(undef, 2); i = 1;
```

```
julia> while i <= 2
```

```
    let i = i
```

```
        Fs[i] = ()->i
```

```
    end
```

```
    global i += 1
```

```
end
```

```
julia> Fs[1]()
```

```
1
```

```
julia> Fs[2]()
```

```
2
```

再查看此时 Fs 的内部结构:

```
julia> dump(Fs)
```

```
Array{Any}((2,))
```

```
 1: getfield(Main, Symbol("###3#4")){Int64}(1) (function of type getfield(Main, Symbol("###3#4")){Int64})
```

```
    #1#i: Int64 1
```

```
 2: getfield(Main, Symbol("###3#4")){Int64}(2) (function of type getfield(Main, Symbol("###3#4")){Int64})
```

```
    #1#i: Int64 2
```

```
julia> Fs[1] == Fs[2]
```

```
false
```

可见, 函数的定义发生了变化, 从 $Fs[1]==Fs[2]$ 的两次结果来看, 函数的表达更新了。

例中的代码片段有:

```
let i = i
```

```
    Fs[i] = ()->i
```

```
end
```


其中, `let i=i` 执行时, 右侧 `i` 是只读的, 只会受到外部 `i+=1` 的影响递增, 而左侧的 `i` 在每次的 `let` 时, 会创建新的内存地址并存入右侧的 `i` 值, 所以 `Fs[i]` 会在 `while` 循环时从 1 遍历到 2; 而在匿名函数 `()->i` 中, `i` 受到 `let` 的影响分别指向了不同的内存, 并存在着不同的值, 所以 `Fs[1]()` 和 `Fs[2]()` 被调用时, 返回了不同的值:

```
julia> i += 3
6
```

```
julia> Fs[1]()
1
```

```
julia> Fs[2]()
2
```

再次改变全局变量 `i` 的值重试, 可以发现 `Fs` 没有受到影响, 因为内部的两个函数指向的内存值与这个 `i` 的内存是不同的。

借助 `let` 的概念, 我们深入地理解一下循环体或推导式在变量域方面的特性。对于这种结构, 其内部变量的内容总会在每次循环中被刷新, 或是说被重新分配了, 例如:

```
julia> Fs = Vector{Any}(undef, 2);
```

```
julia> for j = 1:2
    Fs[j] = ()->j    # j变量每次循环都会不同
end
```

```
julia> Fs[1]()
1
```

```
julia> Fs[2]()
2
```

再例如:

```
julia> function f()
    i = 0
    for i = 1:3      # 此处的i是该层for循环自有的局部变量
    end
    return i        # 返回的实际是函数f中自己定义的局部变量, 不会被for循环修改
end;
```

```
julia> f()
0
```

所以从本质上来说, 可以认为, 循环体或推导式内部的循环语句是被 `let` 结构包围的, 在变量域方面与 `let` 有着相同的特性。

如果需要在循环中使用外部变量, 并希望能够对其进行修改, 则可显式地使用 `outer` 关键字对其进行标识。例如:

```
julia> function f()
    i = 0
    for outer i = 1:3 # 对i进行标识, 表示其为外部的变量, 对其进行的更新修改操作对外部可见
    end
    return i
end
```



```
end;
```

```
julia> f()
```

```
3
```

开发中，我们可以利用 `let` 的特性创建独立的局部域，以实现代码作用域的相互隔离，从而在命名方面及变量控制方面进行有效的管理。

13.4 包

Julia 社区提供了大量的包 (Package) 给开发者使用，所以在我们的开发中，很多功能不需要重新开始，借助已有的包能够节约大量的开发周期及维护成本。官方维护了 Julia 包的目录 (Registry, 项目地址为 github.com/JuliaRegistries/General.git)，登记了已经注册的 (Registered) 包的详细信息 (包括名称、唯一标识 UUID、作者、许可证、代码库地址、版本号以及依赖等)。在网站 pkg.julialang.org 中，能够找到已注册 Julia 包的完整列表，几乎覆盖了各种应用场景，也可以通过官方新增的 juliaobserver.com 网站获得这些包的信息。绝大部分的包都基于 Julia 语言本身开发的，并在著名的开源网站 Github 中托管，所以我们能够看到这些包的源代码，有兴趣的读者可以更深入地了解。

对于注册包，因为信息完整，所以能够很容易地通过 Julia 的包管理器进行安装、更新和卸载等操作。但包管理器并不仅限于注册包，也可以管理未注册包，而且能够协助开发者开发出自己的包或者更新修改现有的包。

13.4.1 管理机制

在对本地安装包的管理方面，v1.0 版的 Julia 提供了全新的包管理器，采用类似于 Python 中 `virtualenv` 的方式，能够针对不同的项目提供独立的环境，可安装不同组合的包，而不再是全局性的包集合。

所谓项目，指的是我们开发过程中为了某个业务或功能而建立的代码集合，通常会有一个根目录，用于存储记录该项目所涉及的所有文件，一般这个目录被称为项目的工作目录或主目录。

我们知道，一个项目的开发有一定的周期性，而开发时总会基于当时的环境设计各种函数、对象和接口等，但在交付及部署时很有可能因为环境的不一致导致项目无法正常运转。或者一段时间后，语言及环境也很有可能已经升级更新，如果采用最新版，项目的依赖一旦在新版中不存在或不再支持，也会导致项目无法正常运行，因为我们无法确保项目的实现一定是向后兼容的，尤其是无法确保外部的环境在变化时仍符合项目运行的要求。所以在维护上，往往我们需要对项目进行描述，对依赖库、接口方式等各方面进行详细的说明，并提供尽可能详尽的安装、部署及使用手册。

而这种基于项目的包或库的管理方式，直接将项目的依赖与环境的配置紧密关联，使



得项目的开发状态有了更为精准的快照，不但能够极大地简化项目的环境信息描述，更能够方便后续部署时快速地复原项目所需的条件，极大地降低了维护成本。

Julia 在发布的 v1.0 版中放弃了之前传统的包管理方式，使用了这种更为符合开发需求的机制，能够根据用户指定的项目主目录进行独立的环境配置，而且针对依赖包的版本控制精确到了 Git 每次提交的 SHA1 值（简称为版本 ID）。

在 Julia 安装之后，第三方包的默认安装目录为：

```
~/.julia/packages/包名/版本ID
```

其中，~ 表示操作系统中的用户目录，Linux 中一般在 /home 中，而 Windows 一般在 C:\Users 中。版本 ID 表示了包的某次更新快照。由此可见，我们可以同时安装某个包的不同版本。例如：~/.julia/packages/DataFrames/QyzTe 便是版本 ID 为 QyzTe 的 DataFrames 包安装路径。

虽然不同的项目可以有不同的包组合（安装了哪些，又是什么版本），但很显然无须对每个项目独立安装这些包，尤其是不用在每个项目都安装一套。包库只需一套即可，而所谓环境快照，实际为各项目建立的依赖描述信息，记载着该项目需要哪些包以及它们的版本信息。

在每个项目的根目录中，都会有两个文件，分别是 Project.toml 或 JuliaProject.toml，以及 Manifest.toml 或 JuliaManifest.toml 文件。一般会在我们第一次为该项目安装包时才会自动创建。

前者记录了项目所依赖的包以及版本 ID，例如：

```
[deps]
CSV = "336ed68f-0bac-5ca0-87d4-7b16caf5d00b"
Example = "7876af07-990d-54b4-ab0e-23690620f79a"
```

表示当前项目依赖两个包，分别是 CSV 和 Example，而每个包的版本 ID 是上述两个很长的字符串。不仅如此，我们还可以在该文件中放入其他有关项目的信息，包括项目名称、作者、许可证以及本项目的 UUID 等。

后者描述了项目当前的 Julia 开发环境中所有可用的模块以及包信息，内容类似于：

```
[[LinearAlgebra]]
deps = ["Libdl"]
uuid = "37e2e46d-f89d-539d-b4ee-838fcccc9c8e"

[[Markdown]]
deps = ["Base64"]
uuid = "d6f4376e-aef5-505a-96c1-9c027394607a"

[[Random]]
deps = ["Serialization"]
uuid = "9a3f8284-a2c9-5f02-9a11-845980a1fd5c"

[[Serialization]]
uuid = "9e88b42a-f829-5b0c-bbe9-9e923198166b"

[[Sockets]]
```




```

uuid = "6462fe0b-24de-5631-8697-dd941f90decc"

[[CSV]]
deps = ["CategoricalArrays", "DataFrames", "DataStreams", "Dates", "InternedStrings",
        "Missings", "Mmap", "Test", "WeakRefStrings"]
git-tree-sha1 = "38fb8a40079560da5804a280d7d3cf0180b833b1"
uuid = "336ed68f-0bac-5ca0-87d4-7b16caf5d00b"
version = "0.3.0"

[[Example]]
deps = ["Test"]
git-tree-sha1 = "8eb7b4d4ca487caade9ba3e85932e28ce6d6e1f8"
uuid = "7876af07-990d-54b4-ab0e-23690620f79a"
version = "0.5.1"

```

其中包括各个模块的依赖包列表、Git 版本 ID、包的 UUID（用于准确地标识一个包）以及版本号等，例如上述安装的 CSV 包的版本是 v0.3.0，而其依赖的包有 DataFrames、Dates 及 Missings 等。



提示 在本地安装环境中，无论是项目路径中的 toml 文件还是 packages 目录，都是 Julia 自动维护的，若无必要，最好不要手动修改，以免引起混乱。

为了方便针对项目进行包依赖管理，Julia 提供了一种专门的 Shell 模式（为叙述方便，将其简称为 PkgREPL）。只需在 REPL 的 `julia>` 提示符下，按下右方括号，即可开启该模式，出现类似如下的提示符：

```
(v1.0) pkg>
```

圆括号内展示了当前项目的根目录，默认为 `~/.julia/environments/v1.0`（针对 v1.0 的 Julia）。在该提示符下，输入简单的命令便可实现包的安装、卸载以及更新等功能。如果要退出该模式，只需按下删除键（BackSpace 键）或 CTRL^C 便会退回到 `julia>` 提示符下。

在 `pkg>` 下输入 `help` 命令，可以获得 PkgREPL 下能够使用的各种命令列表，如下所示：

```

status: 展示项目环境的基本信息
add: 为项目安装包
develop: 将某个包的代码克隆到本地以便于开发
rm: 从manifest或项目中移除包
up: 更新manifest文件中变更的包
test: 运行包的测试用例
build: 运行包的构建脚本
pin: 固化包的版本以禁止其被更新
free: 取消固化、开发等状态，回到注册版本
instantiate: 下载项目的所有依赖
resolve: 更新manifest以确保与依赖变更的一致性
generate: 为新项目生成相关文件
preview: 预览某个命令的执行过程，但不会对当前环境及项目的状态做出实质性的修改
precompile: 预编译项目的所有依赖包
gc: 自动清理项目不再需要的包
activate: 设置包管理器操作的主目录（一般为项目的根目录）

```



上述这些命令覆盖了下载、安装、测试、构建、更新、卸载及清理各种功能，但需要在 PkgREPL 下执行。当然，目前的 v1.0 仍保留了旧式的管理方式，即通过模块 Pkg 的 API 执行上述各种功能。例如，在 julia> 提示符下执行 `Pkg.add("包名称")` 便可安装某个包，或者执行 `Pkg.rm("包名称")` 删除某个包。



注意 Julia 对包的名称是大小写敏感的，所以应提供准确的信息。

下面的内容我们以最新的 PkgREPL 模式介绍包的管理功能，API 方式可参阅官方文档。

13.4.2 安装移除

1. 工作目录激活

假设我们现在需要使用 Julia 语言开发某个项目 MyProject，其主目录是同名的文件夹（假设完整路径为 `d:\dev\MyProject`）。此时该文件夹内是空的。

在做好规划后，便需要配置开发环境了。

首先，切换到 PkgREPL 模式，注意圆括号内的项目路径是否为我们的项目主目录，如果不是，则需要通过将 PkgREPL 的操作目录激活为我们的项目目录 MyProject，即：

```
(v1.0) pkg> activate d:/dev/MyProject
```

该命令指定了 PkgREPL 后续操作的对象（即项目根目录），路径参数可以是相对路径也可以是绝对路径。



注意 Windows 下输入路径时可使用斜杠表达路径，避免 Julia 解析的问题。

在执行成功后，提示符应变化为：

```
(MyProject) pkg>
```

再次提醒，请注意圆括号中的名称，确保是我们需要操作的项目根目录。不过此时的 MyProject 内仍是空的。

可以认为 `activate` 命令仅仅切换了 PkgREPL 的工作目录，不会再做更多的事情。

2. 安装注册包

为一个项目安装注册包是非常简单的，只需在 PkgREPL 下执行以下命令即可：

```
(项目目录) pkg> add 包1 包2 包3 ...
```

每次执行可以同时安装多个包，各包的名称使用空格分开。例如：

```
(MyProject) pkg> add Example
Updating registry at `~\.julia\registries\General`
Updating git-repo `https://github.com/JuliaRegistries/General.git`
Resolving package versions...
Updating `d:\dev\MyProject\Project.toml`
[7876af07] + Example v0.5.1 # 绿色字体，表示增加的内容
```



```
Updating `d:\dev\MyProject\Manifest.toml`
[7876af07] + Example v0.5.1           # 绿色字体, 表示增加的内容
[2a0f44e3] + Base64                   # 绿色字体, 表示增加的内容
[8ba89e20] + Distributed
[b77e0a4c] + InteractiveUtils
[8f399da3] + Libdl
[37e2e46d] + LinearAlgebra
[56ddb016] + Logging
[d6f4376e] + Markdown
[9a3f8284] + Random
[9e88b42a] + Serialization
[6462fe0b] + Sockets
[8dfed614] + Test
```

执行时首先会下载 JuliaRegistries/General.git 并更新到本地的 `~\.julia\registries\General` 目录, 以获取官方注册包的最新信息。同时会更新项目 `MyProject` 中的 `Project.toml` 及 `Manifest.toml` 文件, 如果不存在则会创建。安装过程会报告包的信息, 包括简短版的 UUID 值、包名、包版本等, 如果是标准库或既有模块 (例如 `Base64`、`Test` 等), 因为是 Julia 环境自带的, 所以不会出现版本号信息。

此后, 我们便会发现 `MyProject` 目录下不再是空空如也, 而是多出了两个 `toml` 文件。其中的 `Project.toml` 文件此时的内容为:

```
[deps]
Example = "7876af07-990d-54b4-ab0e-23690620f79a"
```

在其 `[deps]` 字段记录了安装包的基本信息, 包括了名称与版本 ID; 而另外一个 `Manifest.toml` 文件记录了更为详细的依赖信息。一般我们无须在操作系统中打开文件查看, 在 `PkgREPL` 中使用 `status` 命令 (或 `st`) 便可以查看这两方面的内容:

```
(MyProject) pkg> status           # 查看Project信息
Status `d:\dev\MyProject\Project.toml`
[7876af07] Example v0.5.1

(MyProject) pkg> st --manifest    # 查看Manifest信息
Status `d:\dev\MyProject\Manifest.toml`
[7876af07] Example v0.5.1
[2a0f44e3] Base64
[8ba89e20] Distributed
[b77e0a4c] InteractiveUtils
[8f399da3] Libdl
[37e2e46d] LinearAlgebra
[56ddb016] Logging
[d6f4376e] Markdown
[9a3f8284] Random
[9e88b42a] Serialization
[6462fe0b] Sockets
[8dfed614] Test
```

如果项目对包的版本有特定要求, 则可以在安装时紧随包名使用 `@` 符号以指定某个版本, 例如:

```
(MyProject) pkg> add Example@0.4
```



```

Resolving package versions...
Installed Example — v0.4.1
Updating `d:\dev\MyProject\Project.toml`
[7876af07] ↓ Example v0.5.1 ? v0.4.1      # 紫色字体, 表示将被替换
Updating `d:\dev\MyProject\Manifest.toml`
[7876af07] ↓ Example v0.5.1 ? v0.4.1      # 紫色字体, 表示将被替换
[2a0f44e3] - Base64                        # 红色字体, 表示不再依赖将被去除
...                                         # 其他已省略

```

通常而言, 一个项目只会需要某个包的一个特定版本, 不会同时依赖其多个版本。所以, 在安装包时, 若不限定版本, 则会安装包的最新版; 若限定版本, 则会在环境快照 Project.toml 中替换掉之前的版本信息, 同时会重新评估依赖性, 一旦某些包不再需要便会一并去除。如上例中我们更换了 Example 的版本后, 再次查看状态, 会发现:

```

(MyProject) pkg> st
Status `d:\dev\MyProject\Project.toml`
[7876af07] Example v0.4.1

(MyProject) pkg> st --manifest
Status `d:\dev\MyProject\Manifest.toml`
[7876af07] Example v0.4.1

```

一是 Project.toml 中 Example 的版本号已经被替换, 而在 Manifest.toml 文件中, 除了 Example 外, 其他的包均被去除。

如果包的某个分支或提交存在一些重大修复 (Hotfix), 但还没有注册到官方的 Registry 中, 也同样能够使用 add 命令对该分支或提交进行跟踪, 只需在包名后使用 # 指定分支或版本 ID 即可。例如:

```

(MyProject) pkg> add Example#master
Cloning git-repo `https://github.com/JuliaLang/Example.jl.git`
Updating git-repo `https://github.com/JuliaLang/Example.jl.git`
Resolving package versions...
Updating `d:\dev\MyProject\Project.toml`
[7876af07] ↑ Example v0.4.1 ⇒ v0.5.1+ #master (https://github.com/JuliaLang/
Example.jl.git)                        # 绿色字体
Updating `d:\dev\MyProject\Manifest.toml`
[7876af07] ↑ Example v0.4.1 ⇒ v0.5.1+ #master (https://github.com/JuliaLang/
Example.jl.git)                        # 绿色字体
[2a0f44e3] + Base64                    # 绿色字体, 以下这些包被重新增加
[8ba89e20] + Distributed
...                                     # 其他已省略

```

此时, 项目状态为:

```

(MyProject) pkg> st
Status `d:\dev\MyProject\Project.toml`
[7876af07] Example v0.5.1+ #master (https://github.com/JuliaLang/Example.
jl.git)                                # 附加了master标识

(MyProject) pkg> st --manifest
Status `d:\dev\MyProject\Manifest.toml`
[7876af07] Example v0.5.1+ #master (https://github.com/JuliaLang/Example.
jl.git)                                # 附加了master标识

```




```
[2a0f44e3] Base64
[8ba89e20] Distributed
... # 其他已省略
```

上述的信息显示，项目正在跟踪 Example 包的 master 分支，当更新该包时，会 pull 该分支的代码更新。如果希望不再如此，而要重新追踪 Registry 中注册的版本，可对该包执行 free 命令以释放跟踪要求，即：

```
(MyProject) pkg> free Example
Resolving package versions...
Updating `d:\dev\MyProject\Project.toml`
[7876af07] ↓ Example v0.5.1+ #master (https://github.com/JuliaLang/Example.
jl.git) ⇒ v0.5.1 # 紫色
Updating `d:\dev\MyProject\Manifest.toml`
[7876af07] ↓ Example v0.5.1+ #master (https://github.com/JuliaLang/Example.
jl.git) ⇒ v0.5.1 # 紫色
```

可以看到，Example 的版本又被更换到了 v0.5.1 版，⇒ 之后的 master 标识消失了。

3. 安装非注册包

PkgREPL 不但支持注册包，也支持在 Registry 中不存在的、未注册包（不过需要该包的版本管理库是基于 Git 的）。例如我们在 MyProject 中安装本书的示例代码库，如下所示：

```
MyProject) pkg> add https://gitee.com/juliaproj/bookexamples.git
Cloning git-repo `https://gitee.com/juliaproj/bookexamples.git`
Updating git-repo `https://gitee.com/juliaproj/bookexamples.git`
[ Info: Assigning UUID 5070ddb-8c85-53e1-8e89-0fa7c10719c8 to bookexamples
# 自动分配UUID
Resolving package versions...
Updating `d:\dev\MyProject\Project.toml`
[5070ddb] + bookexamples v0.0.0 #master (https://gitee.com/juliaproj/book-
examples.git) # 绿色
Updating `d:\dev\MyProject\Manifest.toml`
[5070ddb] + bookexamples v0.0.0 #master (https://gitee.com/juliaproj/book-
examples.git) # 绿色

(MyProject) pkg> st
Status `d:\dev\MyProject\Project.toml`
[7876af07] Example v0.5.1
→ [5070ddb] bookexamples v0.0.0 #master (https://gitee.com/juliaproj/bookexamples.git)

(MyProject) pkg> st --manifest
Status `d:\dev\MyProject\Manifest.toml`
[7876af07] Example v0.5.1
→ [5070ddb] bookexamples v0.0.0 #master (https://gitee.com/juliaproj/bookexamples.git)
[2a0f44e3] Base64
[8ba89e20] Distributed
... # 已经省略
```

因为该包并没有采用 Julia 最新的项目结构，所以并没有 UUID 等信息，在安装时 Pkg 会自动为其分配一个 UUID 以便于标识。此时，在 Manifest.toml 中关于该包的信息为：

```
[[bookexamples]]
```




```

deps = ["Distributed"] # 自动发现了依赖包
git-tree-sha1 = "c9470c08b91d83b20ac1c424efdbcd3dc807f151"
repo-rev = "master"
repo-url = "https://gitee.com/juliaproj/bookexamples.git"
uuid = "5070ddb8-8c85-53e1-8e89-0fa7c10719c8"
version = "0.0.0"

```

实际上, 如果采用了 Julia 支持的项目结构, add 命令同时也会将其依赖的其他包一并安装。

此外, Pkg 还支持对本地包的安装, 具体情况可参考官方资料, 这里不再介绍。

4. 移除包

不需要某个包时, 执行 rm 命令即可, 例如:

```

(MyProject) pkg> rm bookexamples
Updating `d:\dev\MyProject\Project.toml`
[5070ddb8] - bookexamples v0.0.0 #master (https://gitee.com/juliaproj/bookexamples.git) # 红色
Updating `d:\dev\MyProject\Manifest.toml`
[5070ddb8] - bookexamples v0.0.0 #master (https://gitee.com/juliaproj/bookexamples.git) # 红色

(MyProject) pkg> rm Example
Updating `d:\dev\MyProject\Project.toml`
[7876af07] - Example v0.5.1 # 红色
Updating `d:\dev\MyProject\Manifest.toml`
[7876af07] - Example v0.5.1 # 红色
[2a0f44e3] - Base64
... # 已省略

```

也可以使用 pkg> rm --manifest DepPkg 以移除某个名为 DepPkg 依赖包, 但需要注意的是, 该命令也会同时移除依赖于 DepPkg 的所有其他包。

不过, 该命令仅仅是修改了项目 MyProject 中记录的依赖信息, 并不会真正地卸载目录 ~/.julia/packages 中包的代码。在项目环境中移除了不需要的包之后, 如果要彻底删除包的代码, 可以使用 Pkg 的 gc 命令进行清理, 例如:

```

(MyProject) pkg> gc
Active manifests at:
`d:\dev\MyProject\Manifest.toml`
`~\.julia\environments\v1.0\Manifest.toml`
`d:\dev\juliaproj\Manifest.toml`
Deleted ~\.julia\packages\BinaryProvider\ZbFxn: 136.354 KiB
Deleted ~\.julia\packages\bookexamples\mhkTE: 11.668 KiB
Deleted ~\.julia\packages\CodecZlib\3RErA: 31.903 KiB
Deleted ~\.julia\packages\Example\w7Ltl: 3.006 KiB
Deleted ~\.julia\packages\Example\XHHuy: 5.614 KiB
Deleted ~\.julia\packages\InternedStrings\DZQ5H: 24.295 KiB
Deleted ~\.julia\packages\TranscodingStreams\ODhkh: 177.365 KiB
Deleted 7 package installations : 390.205 KiB

```

Pkg 会记录所有的项目安装日志, 在执行 gc 时内部会查阅该日志以判断哪些项目还存在以及哪些包正被使用, 如果有些包未被使用便会在 gc 执行时删除。



13.4.3 更新固化

1. 更新包

如果要已将已经安装的包更新到注册了的更新版本，可使用 `up` 命令。例如我们希望将项目 `MyProject` 中的 `Example v0.4` 版更新到最新版：

```
(MyProject) pkg> up Example
Updating registry at `~\.julia\registries\General`
Updating git-repo `https://github.com/JuliaRegistries/General.git`
Resolving package versions...
Updating `d:\dev\MyProject\Project.toml`
[7876af07] ↑ Example v0.4.1 ⇒ v0.5.1 # 绿色
Updating `d:\dev\MyProject\Manifest.toml`
[7876af07] ↑ Example v0.4.1 ⇒ v0.5.1 # 绿色
[2a0f44e3] + Base64 # 绿色
[8ba89e20] + Distributed
... # 已经省略
```

在更新时，同时会更新该包所有的依赖包，以保持版本的兼容性。

若是担心大版本的更新变化太大，使得项目无法正常工作，可以限定更新的版本仅限于小版本（版本号的 `Minor` 部分），执行的命令为：

```
(MyProject) pkg> up --minor Example
```

当然，`up` 命令除了 `--minor` 参数外，还有 `--major`、`--patch` 及 `--fixed` 参数，能够对更新过程进行精细化的控制。

关于 `Julia` 在版本号管理方面的规则，可以参见附录 B 中有关常量 `Base.VERSION` 的描述。

2. 版本固化

如果担心某个包的任何版本更新都会带来风险，那么可以使用 `pin` 命令对已经安装版本进行固化处理，这样就不会自动更新。例如：

```
(MyProject) pkg> pin Example
Resolving package versions...
Updating `d:\dev\MyProject\Project.toml`
[7876af07] ~ Example v0.4.1 ⇒ v0.4.1 □
Updating `d:\dev\MyProject\Manifest.toml`
[7876af07] ~ Example v0.4.1 ⇒ v0.4.1 □
```

命令执行成功后，会在固化的版本号之后附加一个 □ “图钉” 符号。如果此时再更新 `Example` 包，会发现没有更新到 `v0.5.1` 版，即：

```
(MyProject) pkg> up Example
Updating registry at `~\.julia\registries\General`
Updating git-repo `https://github.com/JuliaRegistries/General.git`
Resolving package versions...
Updating `d:\dev\MyProject\Project.toml`
[no changes]
Updating `d:\dev\MyProject\Manifest.toml`
[no changes]
```



若要去掉图钉符号，取消版本固化，需对其执行 `free` 命令，即：

```
(MyProject) pkg> free Example
Updating `d:\dev\MyProject\Project.toml`
[7876af07] ~ Example v0.4.1 ⇒ v0.4.1 # 图钉被移除
Updating `d:\dev\MyProject\Manifest.toml`
[7876af07] ~ Example v0.4.1 ⇒ v0.4.1 # 图钉被移除
```

此后，便又可使用 `up` 命令对 `Example` 执行升级操作了。

13.4.4 小结

Julia 包管理器在 v1.0 版前后发生了很大的变化，并采用了完全不同的机制。不但能够更好地支持项目开发，在使用上也更为方便。基于新的机制，我们可以为不同的项目维护独立的运行环境，而且相互不会干扰，对依赖的控制也更为准确。

旧式的 API 管理方式在 v1.0 版中依然可以使用，但不排除今后会被弃用，所以建议开发者采用新的方式对项目的包进行管理。



并行计算

现代计算机早已进入多核时代，强大的计算力已非往日可比。同时，随着互联网所带来的变革，多台计算机更可通过网络连成集群，借助分布式技术无限地扩展其数据处理能力。

迄今为止，无论硬件如何发展，网络如何发达，影响计算力的基础依旧是 CPU 的计算速度和存储装置的存取速度。显而易见，在集群中，对内存区具有最快存取速度的是本地 CPU，因为两者处于同一个节点。同理，离 CPU 更近的缓存比内存具有更快的存取速度。

集群将任务分发到多台机器上，通过分布式机制实现并行处理。如上所述，充分发挥本机的并发能力，是整体性能提升的关键。多进程、多线程等便是挖掘多核优势的常用手段。但是，无论是分布式还是本地的并发处理，都是为了任务的统一目的，所以必然也会涉及协调同步、数据共享及传递等问题。

Julia 在并行计算方面提供了很多强大的机制，包括协程任务、数据通道、远程调用等，本章将一一进行介绍。

14.1 基础概念

在学习 Julia 的并行机制之前，我们先了解一些相关的基础知识。

14.1.1 进程与线程

进程是应用程序的执行实例，是 CPU、内存及各种计算、存取设备等资源分配的最小单位，也是计算调度的基本单位。不过，CPU 与其他外围的资源有着明显的区别：外围设备主要是存取装置、输入输出设备或控制设备；而 CPU 则是数据转换的计算中心，有着远超其他资源的运转性能。所以在多任务、多进程的操作系统中，调度与分配的焦点是 CPU

的计算资源。

微观上看，多任务系统中的进程是轮流切换执行的，它们之间的调度实际上是对 CPU 占用时间片的管理。当某进程用完了 CPU 分配的执行时间片，就会被切换出去等待下次轮换的到来。在切换过程中，为了能够让被切换出去的某个进程下一次进入 CPU 时仍能按着当时的状态继续运行，就需要将该进程的各种资源（包括键盘、鼠标、内存变量、缓存等等现场情况）再次恢复过来，所以切换出去之前就需要保存当前的状态，即所谓的程序上下文。

显然，在切换 CPU 计算资源的同时，来回更新慢速的外围装置的状态不是高效的方式，因为计算逻辑的实现往往是性能的核心焦点。所以为了更合理地利用资源，现代操作系统创建了线程机制将进程并行化，实现了资源分配与计算调度的分离，使得 CPU 调度变得更为纯粹，能够让整体效率得到大幅度提升。

可以认为，进程是资源分配的最小单位，线程则是 CPU 计算能力调度的最小单位。一个进程可以有多个线程，但总会有一个主线程。但线程并非是资源分配单位，只是进程的不同执行路径，共享着同属进程所控制的资源，无法从操作系统直接获得资源的控制权与使用权，所以线程依赖于进程。操作系统为了识别进程与线程，会对每个进程与线程编号，即所谓的进程 ID (PID) 或线程 ID。

有关并行化与分布式的功能一般均在 Julia 的模块 Distributed 中提供。在 Julia 的并行环境中，将进程称为工作者 (Worker)。默认情况下，Julia 启动时只会开启一个 Worker，其 PID 默认为 1；运行中可以通过 `addprocs()` 和 `rmprocs()` 两个函数随时增加或减少 Worker 的数量，此时其他 Worker 的 PID 会依次从 2 开始，并被视为远程 Worker。也可以在启动时以 `julia -p n` 的方式启动多进程的 REPL，其中 `n` 是进程数，通常是 CPU 核数。我们可以随时调用 `procs()` 函数获得所有 Worker 的 PID，包括本地默认的 Worker；也可以通过 `workers()` 查看所有远程 Worker 的 PID。

14.1.2 条件变量

线程的出现大大提高了资源利用率，也提供了程序的并发能力。但在数据处理过程中因这种并发执行的存在，同一份数据处理或读写时便出现了同步与协调问题。解决这种问题的机制之一便是条件变量 (Condition Variable)。

实现中，条件变量通常是被所有线程可见的全局变量，并可改变状态。线程在执行过程中可判断其状态并阻塞等待其状态发生变化；一旦另一线程改变了其状态，当前线程便可及时发现变化事件，恢复运行后面的处理过程。

由于条件变量本身也是一种数据，所以其本身状态的修改也会导致冲突，引发同步问题。所以在开发中，条件变量通常会与互斥量 (Mutex) 结合使用。

互斥量通过加锁 (Lock) 与解锁 (Unlock) 来保护一片被称为临界区的代码段。在运行的任意时刻，临界区总会只有一个线程访问执行，而其他线程需要等待当前线程完成并解锁，才能够进入该临界区。如此一来，该临界区内的赋值、修改等操作便具有独立性，不会同时发生导致冲突。

关于条件变量及互斥量等同步机制的更多内容，请参阅相关资料。

14.2 协程调度

在程序中经常会遇到等待设备是否就绪的情况，典型的场景是进行各种 IO 操作时，例如对磁盘文件的加载读写、访问 Web 服务、调用外部程序等。这种等待过程如果是阻塞的，很容易带来资源的空耗。此时，完全可以将其他的处理过程切换进来，将挂起的计算资源转交给该任务，这样便能够极大提高资源的整体利用率，同时大大提高整体执行效率。

为了能够对一段处理过程方便地暂停、切换、恢复，将其中的操作进行特别的封装，便成了不同于常见的多进程或多线程的并发机制，即协程（Coroutines），在 Julia 中称为任务（Tasks）。为叙述方便，后文中提及协程或 Task 指的是同一个概念。

协程并不是进程或者线程这种系统级别的并发设施，可以看作是用户态的轻量级线程，所以在切换过程中的消耗要比进程、线程上下文切换时要小得多。更为吸引人的是，协程不会涉及让人头疼的同步或锁的问题，因为它不像进程、线程那样是抢占式多任务机制，而是协作式多任务机制。理论上讲，协程比线程的执行效率更高，而且因为占用的额外消耗极少，所以在一个系统中完全可能出现成千上万的协程，远远超出进程或线程的并发程度。

在 Julia 中创建协程 Task 非常简单。构造方法 `Task(f::Function)` 与 `@task` 宏分别将一个函数对象与一个表达式封装为协程 Task 对象；但是函数参数 `f` 必须能够进行无参调用（没有参数或所有参数都有默认值）。

例如，分别以这两种方式创建 Task 对象，用于将数组的所有元素修改为单一的值，如下：

```
julia> a = zeros(1, 5)           # 一个初始值均为0的数组a
1×5 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0
```

```
julia> f() = fill!(a, 1.0)      # 定义一个无参函数对象f
f (generic function with 1 method)
```

```
julia> tsk1 = Task(f)           # 将f封装为Task对象
Task (runnable) @0x0000000023931cd0
```

```
julia> tsk2 = @task fill!(a, 2.0) # 使用宏直接将操作表达式封装为Task对象
Task (runnable) @0x0000000023931fb0
```

在协程对象在创建之初，被封装的操作不会立即执行。此时的数组 `a` 仍为初始状态，即：

```
julia> a                       # 至此数组a没有任何变化
1×5 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0
```

但在创建 Task 时，下述的方法是错误的：

```
# 参数表达式会在传入时进行求值运行，所以传入的参数并不是函数对象
```



```

1.0 1.0 1.0 1.0 1.0

julia> schedule(tsk2)           # 启动另外一个Task, 并完成
Task (done) @0x0000000023931fb0

julia> a                       # 数组a的内容被修改为新的值
1×5 Array{Float64,2}:
 2.0 2.0 2.0 2.0 2.0

```

可见, 在将两个 Task 对象加入调度系统时, 它们会启动生效。当其中的处理完成后, 该 Task 也同时结束 (done 状态)。

此时再查看它们的状态, 会发现两者都已启动且已经完成, 即:

```

julia> istaskstarted(tsk1)
true

julia> istaskstarted(tsk2)
true

julia> istaskdone(tsk1)
true

julia> istaskdone(tsk2)
true

```

为了便于使用, Julia 提供了另外一个与 `schedule()` 函数相当的宏 `@async`, 可以在将一个表达式创建为 Task 对象的同时将其加入到调度队列中。例如:

```

julia> using Distributed

julia> @async fill!(a, 4.0)
Task (done) @0x0000000023933990

julia> a
1×5 Array{Float64,2}:
 4.0 4.0 4.0 4.0 4.0

```

可见, `@async` 直接将 `fill!(a, 4.0)` 创建为 Task 时, 便加入到队列中并启动运行了。

为了能够更深刻地理解协程的作用, 再来看一个复杂点的例子。

假设在脚本 `demo_sleep.jl` 中写入如下的代码:

```

function f()                     # 函数f, 期间会sleep两次
    println("f starting...")
    println("f sleep 1...")
    sleep(1)                     # 1秒
    println("f sleep 2...")
    sleep(2)                     # 2秒
    println("f wake up 2.")
    println("f finished.")
end

function g()                     # 函数g, 也会sleep两次, 但周期与f不一样
    println("g starting...")
    println("g sleep 1...")
    sleep(2)                     # 2秒

```



```

println("g sleep 2...")
sleep(0.3)           # 0.3秒
println("g wake up 2.")
println("g finished.")
end

t1 = Task(f)          # 将两个函数封装为Task对象
t2 = Task(g)

schedule(t1)          # 加入到调度队列中
schedule(t2)

read(stdin, Char)     # 与演示功能无关，仅用于阻塞脚本避免上述的异步操作过早退出。

```

其中，定义了两个函数，分别会 `sleep` 两次且时间间隔不相同。在将它们封装为 `Task` 对象并加入到调度队列后，运行结果如下：

```

f starting...
f sleep 1...          # f进入sleep
g starting...         # g被切入执行
g sleep 1...          # g进入sleep
f sleep 2...          # 切换到f
g sleep 2...          # f sleep切换到g
g wake up 2.
g finished.           # g执行完切换到f函数
f wake up 2.
f finished.

```

从运行的信息可见，在 `f()` 或 `g()` 进入 `sleep` 时，队列会自动切换到另外一个 `Task` 执行，不会让整个队列停顿阻塞，高效地利用了计算资源。

在 Julia 的协程机制中，除了在 `sleep()` 等事件时让调度系统自动切换 `Task` 外，也可以主动进行切换。例如在脚本 `demo_yield.jl` 中有如下代码：

```

function f()
    println("f starting...")
    println("f pause 1...")
    yield()           # 要求切换到别的Task
    println("f back 1 and pause 2...")
    yield()           # 要求切换到别的Task
    println("f back.")
    println("f finished.")
end

function g()
    println("g starting...")
    println("g pause 1...")
    yield()           # 要求切换到别的Task
    println("g back 1 and pause 2...")
    yield()           # 要求切换到别的Task
    println("g back.")
    println("g finished.")
end

t1 = Task(f)

```

```
t2 = Task(g)
```

```
schedule(t1)
```

```
schedule(t2)
```

```
read(stdin, Char)
```

其中的 Task 都会调用两次 `yield()` 函数，执行的结果为：

```
f starting...
```

```
f pause 1...
```

```
g starting...
```

```
g pause 1...
```

```
f back 1 and pause 2...
```

```
g back 1 and pause 2...
```

```
f back.
```

```
f finished.
```

```
g back.
```

```
g finished.
```

可见，在 `yield()` 调用处会切换到另外一个 Task 执行，然后在另外的 Task 要求切换时，会再次恢复到 `yield()` 处继续执行后续的语句。

上例中的 `yield()` 函数，是调度系统提供的较为底层的功能，用于在 Task 中主动要求于当前位置切换到其他任务中。不过在未提供参数时，调用方不会限定待切换的是哪个 Task。如果暂时没有其他可切换的 Task，当前 Task 会立即重新启动，继续运行。

另外还有一个 `yieldto()` 函数，能够显式地指定要切入哪个 Task。但官方不建议使用该函数，因为其过于底层，在执行切换动作时不会考虑状态及队列秩序（Scheduling）问题，所以也不作过多的介绍。

14.3 数据通道

线程是一种轻量而高效的并发机制，但一般只能在本机上实现多逻辑路线的切换，无法跨机器或在集群中实现并行化。所以，在 Julia 的分布式框架中，多进程是主要的并行方案。

但在这种机制中，进程之间的协同工作难免会涉及消息传递与数据通信。为此，Julia 在其并行框架中引入数据通道（Channel）这一基础设施。通道机制不但能够用于跨进程的通信，同样适用于上文所介绍的 Task 之间的协同。为了便于解释，本节以 Task 为例说明数据通道的原理以及用法。

14.3.1 Channel 对象

Julia 中的 Channel 对象其实是一种带有阻塞特性的先进先出（first-in first-out, FIFO）队列，可以理解作为一种管道（Pipe），可同时被多个任务并发地、安全地读写。其原型为：

```
Channel{T}(sz::Int)
```

若类型 `T` 未指定，则可容纳任意的类型；若参数 `sz>0`，则会建立 `sz` 大小的缓冲区，且

sz 会限定可以容纳的最大元素个数, 但如果 sz 为 0, 则不会建立缓冲区。

例如, Channel(32) 会创建一个最大容纳 32 个任意类型的对象, 而 Channel{Int64}(128) 则创建一个对象, 其最多容纳 128 个类型为 Int64 的元素。

在 Channel 对象创建后, 便可分别通过 take!() 和 put!() 函数对其进行读写操作。例如:

```
julia> c = Channel{2}
Channel{Any}(sz_max:2,sz_curr:0)

julia> x1 = put!(c, 1)
1

julia> x2 = put!(c, 2)
2

julia> y1 = take!(c)
1

julia> y2 = take!(c)
2
```

可见, put!() 操作在放入元素时同时会返回该元素, take!() 操作会按放入的先后顺序逐个提取。

如果在元素提取完再次执行 take!(), 或者放满元素后再次执行 put!() 操作, 会发现这两个操作都没有立即返回, 而是出现了阻塞等待状态。事实上, 每次 take!() 函数被调用时, 只会从 Channel 中提取一个元素同时移除该元素; 但如果 Channel 为空, 该函数阻塞直到有新的元素加入其中。相对地, 函数 put!() 会将一个元素放入 Channel 对象中, 但如果队列已满, 该函数会一直阻塞直到 Channel 中某个元素被移除并有空余的空间。

如果 Channel 没有缓冲区, put!() 的阻塞行为会在 take!() 后立即被释放, 而 take!() 的阻塞会在 put!() 执行后立即被释放。

我们在脚本 demo_ch1.jl 中写入如下的代码, 演示这两个操作的用法以及阻塞时的行为:

```
using Distributed

function putter(c::Channel)      # 往通道c填数据
    println("putting...")
    for i = 1:5                  # 总计填5个元素
        put!(c, i)              # 执行填入操作
        t = rand()
        println("Put ", i, "\t sleep ", t)
        sleep(t)                # sleep随机的t秒
    end
end

function taker(c::Channel)
    println("taking...")
    while true
        x = 0
    end
end
```

```

    t = @elapsed x = take!(c)    # 提取元素并记录操作耗时
    println("Take ", x, "\t elapsed ", t)
end
end

c = Channel{2}                  # 创建一个大小为2的缓冲区
@async taker(c)                 # 异步地执行提取操作
putter(c)                       # 为避免Task之间在sleep等处的自动切换, 故同步地执行放入操作

read(stdin, Char)              # 与演示功能无关, 仅用于阻塞脚本避免上述的异步操作过早退出

```

其中, `putter()` 会间隔随机的秒数在 `Channel` 中放入元素, 而 `taker()` 则会持续地提取内容。在命令行执行该脚本后, 会输出类似如下的结果 (因随机数的不同输出每次都会不同):

```

putting...
Put 1      sleep 0.8379184451921895
taking...
Take 1     elapsed 3.95e-7
Put 2      sleep 0.6438895150182309
Take 2     elapsed 0.836418636
Put 3      sleep 0.9900129382572256
Take 3     elapsed 0.639175616
Put 4      sleep 0.9592257004954332
Take 4     elapsed 0.988817769
Put 5      sleep 0.2108610652543743
Take 5     elapsed 0.959512857

```

可见, 每次 `take!()` 操作并非是立即返回的, 而是都有耗时。这是因为元素是逐一放入的, 而且相隔了一定时间, 导致 `take!()` 操作总会阻塞等待。

在操作 `Channel` 对象时, 如果不希望删除取得的元素, 可使用 `fetch()` 函数, 例如:

```

julia> c = Channel{2}
Channel{Any}(sz_max:2,sz_curr:0)

julia> put!(c, 1);

julia> put!(c, 2);

julia> fetch(c)
1

julia> fetch(c)
1

```

可见 `fetch()` 总会取得队首 (尾) 的元素, 但因未删除所以重复的调用都会返回同一结果。`fetch()` 操作同样会在 `Channel` 为空时发生阻塞, 而且不适用于无缓冲的 `Channel` 对象。

如果我们不希望在提取元素时发生阻塞, 可以利用 Julia 提供的 `isready()` 函数, 先判断 `Channel` 是否有数据存在, 因为 `isready()` 不会阻塞, 因此会立即返回 `Channel` 的状态。例如:


```
julia> c = Channel{2}
Channel{Any}(sz_max:2,sz_curr:0)

julia> isready(c)      # 刚创建时无元素，故立即返回false
false

julia> put!(c, 1);

julia> isready(c)      # 其中有元素，所以为true
true
```

我们可以在 Channel 没有准备好时继续执行其他操作，待其有元素时再执行提取操作；或者可通过函数 `wait()` 等待 Channel 中有可用的元素加入。

但需注意的是，如果通过 `close()` 函数关闭了 Channel，该函数仍会返回 `true`，即：

```
julia> close(c)

julia> isready(c)
true
```

此时如果进行 `put!()` 操作会抛出异常，例如：

```
julia> put!(c, 3)
ERROR: InvalidStateException("Channel is closed.", :closed)
```

但对 `fetch()` 或 `take!()` 有些特别，即：

```
julia> fetch(c)
1
julia> take!(c)
1
julia> take!(c)
ERROR: InvalidStateException("Channel is closed.", :closed)
```

可见，对关闭的 Channel 仍可调用 `fetch()` 或 `take!()` 取得其中余留的数据，但在全部取完后再取数据时会报错。可以使用 `isopen()` 函数判断 Channel 的状态，避免出现上述的错误。

除了上述的特点外，Channel 还是可迭代的，可用 `for` 循环结构遍历其中的元素。例如：

```
julia> c = Channel{2}
Channel{Any}(sz_max:2,sz_curr:0)

julia> put!(c, 10);

julia> put!(c, 11);

julia> for x in c
    print(x, " ")
end
10 11      # 阻塞等待
```

需要注意的是，这种遍历相当于执行了 `take!()` 操作，会“消费” Channel 中的元素。一旦其中的元素被取完，`for` 结构便会阻塞，等待新元素的加入。

我们采用 for 循环方式修改脚本 demo_ch1.jl 后, 在 demo_ch2.jl 中写入新的代码, 即:

```
using Distributed

function putter(c::Channel)
    println("putting...")
    for i = 1:5                # 放入5个元素
        put!(c, i)
    end
    println("closing...")
    sleep(1)                  # 故意sleep 1秒
    close(c)                  # 关闭Channel对象
    println("put over.")
end

function taker(c::Channel)
    println("taking...")
    for x in c                # 采用for循环的方式提取元素
        print(x, " ")
    end
    println("all is taken.")  # 如果for阻塞, 则不会执行该语句
end

c = Channel{2}

@async taker(c)
@async putter(c)

read(stdin, Char)
```

在 putter() 放完 5 个元素后, 通过 close() 函数关闭 Channel 对象; 而 for 循环会持续读取内容。执行该脚本, 结果为:

```
taking...
putting...
1 2 3 4 5 closing...
put over.
all is taken.
```

仔细对照实现的代码可见, 在最后一个元素 5 打印后, for 循环之后的 “all is taken.” 并没有立即执行, 因为 for 结构阻塞了。在 putter() 关闭 Channel 对象并输出 “put over.” 之后, for 循环才释放阻塞并打印之后的信息 “all is taken.” 从而结束整个元素提取过程。其中, close() 函数将 Channel 对象关闭的同时, 也会中断 put!() 与 take!() 操作的阻塞等待状态。

基于 Channel, 我们很容易编写出一个生产者多个消费者的应用。为了便于演示, 在脚本 demo_mconsumer.jl 中输入如下的代码:

```
using Distributed

const jobs = Channel{Int}(32);
const results = Channel{Tuple}(32);
```

```

function do_work()                                # 消费者
    for job_id in jobs                            # 用for循环从输入通道jobs中读取任务数据
        exec_time = rand()
        sleep(exec_time)                        # 用sleep模拟处理过程
        put!(results, (job_id, exec_time))      # 将结果放入输出通道results中
    end
end

function make_jobs(n)                             # 生产者
    for i in 1:n
        put!(jobs, i)                          # 往任务通道jobs中放入数据
    end
end

n = 12;

@async make_jobs(n);                             # 将生产者加入任务调度中

for i in 1:4
    @async do_work()                            # 异步地启动4个消费者，实现并行消费
end

@elapsed while n > 0                             # 打印消费者执行后的结果
    job_id, exec_time = take!(results)          # 从结果通道中提取数据
    println("$job_id finished in $(round(exec_time, digits=2)) seconds")
    n = n - 1
end

```

其中，有 4 个消费者会并行地消费 jobs 通道提供的任务数据，然后将结果并发地输出到结果通道 results 中。执行后会得到类似如下的结果：

```

4 finished in 0.26 seconds
1 finished in 0.27 seconds
6 finished in 0.27 seconds
3 finished in 0.55 seconds
7 finished in 0.32 seconds
2 finished in 0.97 seconds
5 finished in 0.72 seconds
8 finished in 0.51 seconds
11 finished in 0.13 seconds
9 finished in 0.42 seconds
10 finished in 0.51 seconds
12 finished in 0.72 seconds

```

可见，4 个消费者在执行顺序上是任意的、无序的。

14.3.2 通道绑定

根据前文内容，Channel 可用于不同 Task、函数等独立逻辑线的数据交换。而且由于内部的同步机制，同时并发地对其读写也是安全的。这样的结构在常见的生产者 / 消费者问题中会非常有用。事实上，上节中脚本 demo_ch1.jl 与 demo_ch2.jl 中的实现便是生产者 / 消费者问题的典型过程。

但是本节以另外一种方式实现这样的应用。首先定义一个生产者，如下：

```
julia> function producer(c::Channel)
    put!(c, "start")
    for n=1:4
        put!(c, 2n)
    end
    put!(c, "stop")
end;
```

该生产者会在头尾写入“start”和“stop”，而中间则写入4个偶数。

然后使用 for 循环迭代地执行如下代码：

```
julia> for x in Channel(producer)      # 以函数producer构造Channel对象
    println(x)
end
start
2
4
6
8
stop
```

我们会发现，在上例运行时，虽然没有显式地关闭 Channel，但 producer 运行结束后 for 循环并没有阻塞等待，而是正常地结束了。

实际上，其中的 Channel(producer) 是一种特殊的用法，能在创建 Channel 的同时将其与函数 producer 建立绑定关系。在绑定时，producer 会被封装为 Task 对象，而 Channel 对象与其建立关联后，会有共同的生命期。当 Task 随着 producer 运行结束而终止时，Channel 对象也会随之自动关闭。不过此过程要求提供的 producer 函数的定义中只有 Channel 参数。

当然，我们也可以显式地实现这样的过程。先了解一个将 Task 与 Channel 绑定在一起的函数，其原型为：

```
bind(chnl::Channel, task::Task)
```

该函数调用后，会将 chnl 与 task 建立关联，并让它们拥有共同的生命期。例如：

```
julia> c = Channel(0);
julia> task = @async foreach(i->put!(c, i), 1:4);
julia> bind(c, task);
julia> for i in c
    @show i
end;
i = 1
i = 2
i = 3
i = 4
```

其中，Channel 是单独定义的，生产者也被独立地封装为 Task 对象；通过 bind() 将它们绑定在一起后，便可在提取该 Channel 中的内容时获得 task 提供的元素。

此外，该函数还能够重复调用，将多个 Task 绑定到同一个 Channel 上。利用这点，能实现多个生产者对一个消费者的应用。例如：

```
julia> c = Channel{0};

julia> task1 = @async foreach(i->put!(c, i), 1:4);

julia> task2 = @async foreach(i->put!(c, i), 7:9);

julia> bind(c, task1);

julia> bind(c, task2);

julia> for i in c
    @show i
end
i = 1
i = 7
i = 2
i = 8
i = 3
i = 9
```

可见，作为消费者的 for 循环同时取得了生产者 task1 与 task2 提供的内容。



注意 这种情况下只要其中任一 Task 对象结束，被绑定的 Channel 便会关闭。当然，Channel 也可以显式地通过 close() 函数关闭，不必等到 Task 结束时自动关闭。这样的操作需要小心，避免引发不可预料的问题——绑定运行期间，如果 Task 中出现了未捕捉的异常，会通过 Channel 对象传播到所有的消费者。

14.4 远程调用与远程引用

Julia 基于消息传递 (Message Passing) 提供的多重处理环境，允许程序在独立的内存域内同时控制多任务并发执行。但与包括消息传递接口 (Message Passing Interface, MPI) 在内的其他并发环境有所不同，Julia 中的消息通信是单边的。这意味着在 Julia 中，多重的并发处理之间，并非是典型的消息发送与消息接收，而是类似于调用用户函数那样的更高次的操作。开发人员在多处理操作中，只需显式地管理其中一个进程。

Julia 中的并行编程基于两大基础：远程引用 (Remote References) 与远程调用 (Remote Calls)。远程引用是一个可以在任意处理 (进程) 中使用的对象，其指向存储于一特定处理进程中的某个远程对象；而远程调用，则指的是一个处理进程发起的请求，用于以一定参数调用其他处理进程 (或许是自身) 中某个许可的函数。

因为本章介绍的内容主要是跨进程的并行计算方法，需要多个 Worker 参与，所以建议在学习本章前，通过 `julia -p <n>` 或 `addprocs()` 在环境中增加更多的远端进程。为了叙述方便，后文在不必要的情况下，不区分 Worker 与进程这两个概念。

1. remotecall 和 Future

所谓远程调用，指通过本地进程在远程 Worker 中启动某一处理过程。如前所述，不同的进程是调度的基本单位，控制着独立的资源，所以远程调用能够利用本进程之外的其他计算资源实现并行化处理，从而充分发挥多核多机的优势。

在 Julia 中，远程调用的函数原型为：

```
remotecall(f::Function, pid::Integer, args...; kwargs...) -> Future
```

其中，函数对象 f 是要在远程 Worker 中（PID 为 pid ）启动的远程任务；参数 $args...$ 和 $kwargs...$ 用于接收函数 f 的参数。

实施远程调用的 `remotecall()` 为异步函数，其在远程启动任务 f 后，不会阻塞在调用处等待其返回，而是直接运行后面的代码。虽然不会返回函数 f 的执行结果，但 `remotecall()` 在执行时会立即返回一个 `Future` 类型的对象，用于在后续的处理中接收该远程调用的结果。

类型 `Future` 的定义原型为：

```
Future(pid::Integer=myid())
```

其中，参数 pid 默认为当前进程，其内部结构为：

```
Future <: Distributed.AbstractRemoteRef
  where::Int64
  whence::Int64
  id::Int64
  v::Nullable{Any}
```

之所以 `Future` 能够在两个 Worker 中传递结果数据，因为其同时在 `where` 及 `whence` 字段中分别记录了被调用的远程 Worker PID 及调用者的 Worker PID（一般为本地进程），并在 `Nullable` 结构 v 中预留了远端的结果。

例如，通过 `remotecall` 在远程创建一个数组，并通过 `Future` 返回到当前 Worker，如下：

```
julia> using Distributed
```

```
julia> addprocs(4)
```

```
3-element Array{Int64,1}:
```

```
2
```

```
3
```

```
4
```

```
julia> r = remotecall(rand, 2, 2,3)
```

```
Future{2, 1, 4, nothing}
```

在 pid 为 2 的 Worker 中启动 `rand()` 函数，生成 2×3 的随机数组。其返回的 `Future` 对象会立即返回，此时其内部结构为：

```
julia> dump(r)
```

```
Future
```

```
  where: Int64 2
```

```
whence: Int64 1
id: Int64 4
v: Nothing nothing
```

该 Future 对象的 id 被自动分配为 4；当前 Worker 的 PID 为 1，而远端 Worker 的 PID 便是 remotecall() 指定的第二个参数。

从上述的内容也可看出，此时该对象中的 v 字段中并没有获得 Worker 2 中 rand() 的结果。这是因为 remotecall() 是异步执行的，该 Future 对象在被其返回时只是建立了调用者、被调用者及处理过程的关联，不会立即获得处理过程的结果。

如果要取得远程结果，需要使用 fetch() 方法，如下：

```
julia> fetch(r)
2×3 Array{Float64,2}:
 0.61375  0.869206  0.198916
 0.456217 0.0421449 0.64646
```

此时再查看 Future 对象 r 中的结构：

```
julia> dump(r)
Future
  where: Int64 2
  whence: Int64 1
  id: Int64 4
  v: Some{Any}
    value: Array{Float64}((2, 3)) [0.61375 0.869206 0.198916; 0.456217 0.0421449 0.64646]
```

可见，v 字段获得了调用的结果。但如果处理过程出现问题抛出了异常，在 fetch() 远程结果时会上报 RemoteException 错误。

在远程调用时，如果处理过程是极耗时的，在使用 fetch() 时未必能及时获得有效的结果，会一直阻塞等待。为此，可以使用 isready() 函数判断 Future 对象中的数据是否可用，该函数不会阻塞，而是立即返回 Future 对象的有效性。例如：

```
julia> isready(r)
true
```

若要等待远程计算完成，可以使用 wait() 函数。该函数会阻塞，直到等 Future 对象 r 中出现有效的结果，才会结束阻塞状态，然后可以通过 fetch() 取得远程结果。

在处理结果准备就绪后，一旦被 fetch()，远程的结果便会经由远程（网络）传输到本地 Worker 的 Future 对象中缓存起来，而远程的数据会被删除。在本地 Worker 中，可以反复使用 fetch() 调取结果数据，但不会再引发远程传输。例如，上例中生成的随机数组，反复获取时，结果如下：

```
julia> fetch(r)
2×3 Array{Float64,2}:
 0.61375  0.869206  0.198916
 0.456217 0.0421449 0.64646
```

```
julia> fetch(r)
2×3 Array{Float64,2}:
 0.61375  0.869206  0.198916
```



```
0.456217 0.0421449 0.64646
```

可见,多次取得的数组并没有发生变化,因为远程处理后的结果一直缓存在 `r` 中未有变化。

在远程调用时,有时候因故需要等待其完成才能继续进行,为此会执行与如下类似的语句:

```
result = fetch(remotecall(f::Function, pid::Integer, args...; kwargs...))
```

即在远程调用的同时提取其结果,该命令会阻塞直到结果返回。但是,这种操作可以直接使用 Julia 提供的 `remotecall_fetch()` 函数,效果是相同的,例如:

```
julia> remotecall_fetch(rand, 2, 2, 3)
2×3 Array{Float64,2}:
 0.0488111 0.582174 0.06717
 0.0521406 0.511356 0.102627
```

同样,Julia 中的函数 `remotecall_wait()` 提供了 `wait()` 与 `remotecall()` 结合的用法。

另外,若远程调用不需要返回结果数据,便可舍弃 `Future` 机制,因为这种机制会带来额外的消耗。此时,最好使用 `remote_do()` 函数进行进程调用,其原型为:

```
remote_do(f::Function, pid::Integer, args...; kwargs...) -> nothing
```

可见,除了名字不同及不返回 `Future` 对象外,调用方法与 `remotecall()` 是一致的。因为无 `Future` 传递结果,所以 `remote_do()` 中的任何异常会直接输出到标准错误流 `stderr` 中。

在使用中,两者还有另外一个区别:多次 `remotecall()` 时,其中的函数对象会依序启动;但在 `remote_do()` 中启动的远程函数并不能保证执行的顺序。例如有四个函数需要依次调用执行,实现如下:

```
remote_do(f1, 2);
remotecall(f2, 2);
remote_do(f3, 2);
remotecall(f4, 2);
```

实际上,可以确定 `f4()` 会在 `f2()` 之前被启动,但却无法确保 `f1()` 一定会在 `f3()` 之前启动运行。

2. @spawnat 和 @spawn

为了方便使用远程调用,Julia 提供了功能类似于 `remotecall()` 的两个宏,分别是 `@spawnat` 和 `@spawn`,均返回 `Future` 对象。

第一个宏 `@spawnat` 接收两个参数,基本用法为:

```
@spawnat pid 表达式
```

其中,`pid` 是欲运行处理任务的远端 Worker PID;“表达式”必须是可执行的,被自动创建为闭包 (Closure),并被传送到远端 Worker `pid` 中启动运行。例如:

```
julia> using Distributed

julia> s = @spawnat 2 rand(2,3) # 在Worker 2中创建随机数组
Future{2, 1, 27, nothing}
```



```
julia> fetch(s)
2×3 Array{Float64,2}:
 0.349108  0.0537544  0.822414
 0.65166  0.426374   0.932989
```

其中，在 Worker 2 中启动 `rand()` 函数，创建一个随机数组，并可通过 `Future` 对象获得该数组。再看一个例子：

```
julia> using Distributed

julia> r = @spawnat 3 1 .+ fetch(s) # 在Worker 3中对Worker 2的数组逐元素+1
Future{3, 1, 29, nothing}

julia> fetch(r)
2×3 Array{Float64,2}:
 1.34911  1.05375  1.82241
 1.65166  1.42637  1.93299
```

该次远程调用在 Worker 3 中启动处理过程，而且对 Worker 2 中的数组进行处理，将其中的每个元素执行累加操作。需要明确的是，使用的远程结果，应该是 `fetch()` 之后的结果，而不是 `Future` 对象本身。如上例中，在逐元素累加 1 时，用法是 `1 .+ fetch(r)` 而不是 `1 .+ r`，这是因为 `r` 仅是计算结果的远程引用，并不是直接可用的数据。

第二个宏 `@spawn` 仅接收一个表达式参数，即：

`@spawn`表达式

该宏与 `@spawnat` 及 `remote_call()` 都不相同，会自动选择一个可用的远端 Worker，而不需要用户指定。这种用法在远程调用频繁的代码中会非常有用。

例如，我们可以通过该宏启动 `myid()` 函数，获取该函数运行的 Worker ID，如下：

```
julia> using Distributed

julia> workers()
3-element Array{Int64,1}:
 2
 3
 4

julia> fetch(@spawn myid())
2

julia> fetch(@spawn myid())
3

julia> fetch(@spawn myid())
4

julia> fetch(@spawn myid())
2
```

其中，`workers()` 先列出当前可用的 Worker，然后重复远程原型 `myid()`，可见每次运行都不一致，这是因为宏 `@spawn` 在每次启动时选择了不同的 Worker。

一般而言，在使用远程调用时，我们无须在意处理过程是哪一个 Worker 执行的，重要

的是通过这种异步操作，可以充分地利用计算机多核的并发能力。所以在很多情况下，我们都可以使用宏 `@spawn` 代替 `@spawnat` 或者 `remotecall()` 函数。例如：

```
julia> using Distributed

julia> r2 = @spawn rand(1,3)
Future(2, 1, 14, nothing)

julia> fetch(r2)
1×3 Array{Float64,2}:
 0.789612  0.805723  0.871816

julia> s2 = @spawn 1 .+ fetch(r2)
Future(3, 1, 16, nothing)

julia> fetch(s2)
1×3 Array{Float64,2}:
 1.78961  1.80572  1.87182
```

需要明确的是，`remote_do()` 没有返回机制，是与这三者完全不同的远程调用方式，所以不存在相互替代的关系。

另外，类似于 `remotecall_fetch()` 整合了两个操作，Julia 为了使用的便利，提供了宏 `@fetch` 及 `@fetchfrom`，是 `fetch()` 分别与 `@spawn`、`@spawnat` 宏的整合。例如：

```
julia> using Distributed

julia> @fetchfrom 2 myid()
2

julia> @fetch rand(1,3)
1×3 Array{Float64,2}:
 0.333019  0.705939  0.269938
```

使用这种方式时，不再是异步调用。

一旦任务启动，除了 `wait()` 之外，还可以通过宏 `@sync` 等待任务结束。宏 `@sync` 适用于宏 `@spawn`、宏 `@spawnat`、宏 `@distributed` (后文介绍) 和宏 `@async` 动态封装的处理过程。例如：

```
julia> using Distributed

julia> @spawn sleep(1)
Future(2, 1, 14, nothing)          # 立即返回

julia> @time @sync @spawn sleep(1)
1.011460 seconds (888 allocations: 59.745 KiB)
Future(3, 1, 20, nothing)          # 等待约1秒后返回
```

宏 `@sync` 会等待其后的语句执行完才会返回。需注意的是，该宏不适用于 `remotecall()` 函数及 `remote_do()` 等异步操作。

在远程调用的过程中，不仅仅在 `fetch()` 远程结果中发生了数据的转移。实际上，为了能够在远端 Worker 中进行数据处理，在调用时通常需要将作为输入的变量或对象预先移

动到远端。只不过这样的过程不像 `fetch()` 那样显式地进行，而是由 Julia 自动地、隐式地实现。但是传参方式的不同，会导致这种数据移动存在很大的差异。例如，以下两种不同方式实现了矩阵求幂操作。

方式 1:

```
julia> using Distributed

julia> A = rand(1000,1000);
julia> Bref = @spawn A^2;
[...]          # 其他处理1
julia> fetch(Bref);
```

方式 2:

```
julia> using Distributed

julia> Bref = @spawn rand(1000,1000)^2;
[...]          # 其他处理2
julia> fetch(Bref);
```

虽然两种方式的代码看起来相差不大，但其中 `@spawn` 内部的行为有着很大的区别。

方式 1 中矩阵 `A` 在本地构造，另一远端进程对其平方时，需先将 `A` 发送到该进程；而方式 2 中 `A` 的构造与平方操作均是在同一进程中完成，故比方式 1 转移了更少的数据。

在实际的开发中，经常会遇到类似上例中存在多种选择的情况。出于性能与内存消耗等方面的原因，我们需要从中选择既符合应用需求又能高效运行的方案。仍以上例说明：如果本地需要使用矩阵 `A` 进行后续的处理，但对 `A` 求幂之类的计算极为耗时且远大于跨进程移动操作，则方式 1 是较好的选择；如果本地的后续处理不需要对 `A` 做太多操作，但 `A` 的构造定义却因故极为耗费资源，则方式 2 是较好的方案。

所以，我们需充分地认识 Julia 远程调用与引用的特性，结合实际的开发场景选择出最适合的实现方式。因为稍有不同，Julia 代码的运行效率就会有天壤之别。

3. @everywhere

远程调用的另外一个场景是，一些特定的声明、变量、处理等需要在所有的 Worker 中执行。例如，我们在 REPL 中定义一个如下的函数：

```
julia> function rand2(dims...)
    return 2*rand(dims...)
end
rand2 (generic function with 1 method)
```

之后，在 REPL 中直接调用是没有问题的，即：

```
julia> rand2(2,3)
2×3 Array{Float64,2}:
 0.332675  0.689571  0.870412
 0.849894  0.162637  0.302686
```

但如果通过 `@spawn` 对其远程调用，则会出现错误，如下：

```
julia> using Distributed
```

```
julia> @spawn rand2(2,3)
Future(4, 1, 9, nothing)

julia> fetch(ans)
ERROR: On worker 4:
UndefVarError: #rand2 not defined
```

虽然远程调用命令看似运行成功了，但事实上已经发生了异常，可以在 `fetch` 其结果时发现，该远程调用报出了 `UndefVarError` 错误。

这其实涉及跨进程的作用域问题：Julia 中的 Worker 是不同的进程，而不同的进程控制着独立的计算资源，故它们的工作空间也是独立隔离的。所以某个进程中的类型声明、函数定义等都无法被其他 Worker 直接共享可见。REPL 的工作区默认处于 Worker 1 中，所以 `@spawn` 启动的其他 Worker 无从知晓关于 `rand2()` 的定义。

通过 `remotecall()` 及 `@spawn` 等进行远程调用时，虽然可把本地进程中的数据复制传递到远端 Worker 中，但声明与定义语句不会自动迁移，所以需在其他 Worker 中有一份拷贝。

Julia 提供了一个强大的宏 `@everywhere`，能够实现在所有的远端 Worker 中执行同样的命令。差别仅在于该宏不会返回 `Future` 对象，没有计算结果传递的机制。

接着上例，通过 `@everywhere` 将 `rand2()` 的定义过程在所有其他 Worker 中执行一遍，即：

```
julia> @everywhere function rand2(dims...)
    return 2*rand(dims...)
end
```

如果没有报错，则说明执行成功。然后，再次使用 `@spawn` 在远端调用 `rand2()` 函数：

```
julia> @spawn rand2(2,3)
Future(3, 1, 16, nothing)

julia> fetch(ans)
2×3 Array{Float64,2}:
 1.98105  0.712961  0.783507
 0.648758  0.688808  1.09202
```

可见，成功地在 Worker 3 中创建了随机数组，并可 `fetch` 到本地 Worker 1 中。

再看一个例子：我们通过 `@everywhere` 在所有进程空间中定义一个 `bar` 变量，如下：

```
julia> @everywhere bar=1
```

此后，在其他 Worker 中输出该变量的值，如下：

```
julia> fetch(@spawn myid(), bar)
(2, 1)

julia> fetch(@spawn myid(), bar)
(3, 1)

julia> fetch(@spawn myid(), bar)
(4, 1)
```


可见，在 Worker 2、3、4 中均正确地输出了该变量的值。

如果通过 `@everywhere` 将本地 Worker 中的变量赋值给另外一个变量，则会出错，即：

```
julia> foo = 1;
julia> @everywhere bar=foo
ERROR: On worker 2:
UndefVarError: foo not defined
```

其实道理与上述的相似。这是因为 `foo` 的定义仅在 Worker 1 中有效，对其他的 Worker 并不可见，所以会报出 `UndefVarError` 错误。

为了能够实现这种操作，我们可以借助 `@eval` 宏的扩展能力，即：

```
julia> foo = 1;
julia> @eval @everywhere bar=$foo
```

其中，`@eval` 会在 `@everywhere` 起效之前将表达式中 `$` 标记的变量展开，然后才会对包括 `@everywhere` 在内的表达式进行“求值”操作。

4. @distributed

假设我们要模拟一个投币游戏^①，需要统计无数次投掷中正面与反面出现的次数。其中的投掷函数定义在脚本文件 `count_heads.jl` 中，如下：

```
function count_heads(n)
    c::Int = 0          # 计数器
    for i = 1:n          # 投掷n次
        c += rand{Bool}  # 生成Bool型随机值，隐式转换为Int型，并将true作为1记录到变量c
    end
    c                    # 函数返回统计的true的数量，即为头像一面的数量
end
```

该函数通过随机的布尔值模拟正反面出现的事件，并在 `n` 次的投掷中统计随机布尔值为 `true` 的次数。

为了提高效率，这个过程完全可以通过多进程并行的方式实现，如下：

```
julia> using Distributed

julia> @everywhere include("count_heads.jl")

julia> a = @spawn count_heads(100000000)
Future{2, 1, 6, nothing}

julia> b = @spawn count_heads(100000000)
Future{3, 1, 7, nothing}

julia> fetch(a) + fetch(b)
100001564
```

其中，`@everywhere` 执行 `include()`，将 `count_heads()` 函数的定义加载到其他远端 Worker 中；宏 `@spawn` 自动启动远程 Worker 以启动该函数，再通过 `fetch()` 分别将多个远端进程的执行结果汇总相加，从而得到 2 亿次投掷后，头像一面出现的总次数。

① 这其实是一种最简单的蒙特卡罗仿真，在数理统计、信号处理、数据处理或机器学习中经常会遇到。

投币游戏的实现正如上例所示，是非常简单的。但实践中会遇到大规模的、极为耗时的的问题，比如机器学习中的蒙特卡罗仿真。为了能够在耗时要求严格的情况下，并发地实现该过程，每个远端进程可能无法执行1亿次，而只能执行1万次甚至是100次。此时，很可能需要傻傻地将 `@spawn` 远程调用语句重复地实现很多次，或者需要实现一个循环结构并将总次数进行合理的分配。

实际上，这种在独立分散的并行处理之后再各结果聚合（Combination）的过程，是一种常见的归约（Reduction）处理。例如，汇总高维数组中元素的和或乘积，可以按列或者按行，也可以按线性索引进行。这期间的计算过程，对数据聚合的执行顺序不会影响最终的结果。

对于这种典型的归约问题，Julia 提供了比开发人员自己写分散及聚合过程更好的方法。例如投币游戏可改写为：

```
julia> using Distributed

julia> nheads = @distributed (+) for i = 1:200000000
                                Int(rand{Bool})
                                end
99998665
```

可见，获得了合理的结果，而且代码极为简洁。

上例中的 `@distributed` 宏便是 Julia 提供的基于远程调用的归约计算方式。该宏针对其后的 `for` 循环，能自动启动任意数量的远端 Worker，将循环体的执行过程自动分配到这些 Worker 中。而且 `@distributed` 还可以在其后指定聚合函数，将每个 `for` 循环内的执行结果自动进行归约处理。

就上例而言，`@distributed` 对 2 亿次 `for` 循环自动进行远程调用，并将 `for` 内表达式的结果通过指定的累加运算符进行聚合，得到最后随机值为 `true`（对应整数值 1）的总数量。这种做法显然比前文中的实现方法更为简洁高效。



注意 循环体的执行并不像看起来那样依序执行。正如前面所述，这种归约过程中，每层迭代实际进行了远程调用，会运行在不同的 Worker 中，所以并没有特定的执行顺序。

再看一个例子：如果在远程调用中不是单纯的数据传递，而是希望各进程能够共同操纵同一个内存区的数据，实现并行的修改操作，如下所示：

```
julia> a = zeros(1,5)
1×5 Array{Float64,2}:
 0.0  0.0  0.0  0.0  0.0

julia> @distributed for i = 1:5
                    a[i] = i
                    end
Task (queued) @0x00000000102d5cd0
```

该代码希望将 `a` 中的元素值修改为索引值。执行后会发现，无论我们是否对该数组执行 `fetch.(ans)` 将远端的结果取回，都会发现数组 `a` 仍保持原样，没有任何变化。

实际上不难理解, 因为绑定到变量 `a` 的数组在 `@distributed` 执行远程调用的过程中会传递到其他远程 Worker 中, 重要的问题是这些 Worker 中的同名数组 `a` 是各进程独立控制的, 并非指向同一个内存区, 而是完全隔离的对象。所以, 在 `for` 循环修改数组元素时, 改变的仅是自己 Worker 中的数组, 而不是在本地 Worker 中的同一份数组 `a`。我们可以通过调用 `@spawn` 来查看每个 Worker 中 `a` 的内容, 如下所示:

```
julia> fetch(@spawnat 2 a)
1×5 Array{Float64,2}:
 1.0  2.0  0.0  0.0  0.0

julia> fetch(@spawnat 3 a)
1×5 Array{Float64,2}:
 0.0  0.0  3.0  0.0  0.0

julia> fetch(@spawnat 4 a)
1×5 Array{Float64,2}:
 0.0  0.0  0.0  4.0  0.0

julia> fetch(@spawnat 5 a)
1×5 Array{Float64,2}:
 0.0  0.0  0.0  0.0  5.0
```

可见每个 Worker 都有一个数组 `a`, 而改变的位置正好是 `for` 循环调度到该 Worker 的时候。

此类问题可以通过 Julia 提供的共享数组解决, 请继续阅读后文, 便可获得答案。

5. RemoteChannel

事实上, 前文介绍的 Channel 是 AbstractChannel 的子类型, 属于该类型的一个具体实现; 其相关的操作包括 `put!()`、`take!()`、`fetch()`、`isready()` 和 `wait()` 等也是 AbstractChannel 类型要求必须实现的接口。

但类型 Channel 仅在进程内有效, 无法跨进程使用。举例说明:

```
julia> using Distributed

julia> c = Channel{Any}(2)           # 大小为2的Channel
Channel{Any}(sz_max:2,sz_curr:0)

julia> @fetchfrom 2 put!(c, 10)      # 在Worker 2中往同名对象c中放入元素
10                                   # 成功

julia> isready(c)                    # 但本地的c却仍不是ready的, 说明其中没有任何内容
false                               # 所以在本地进程中如果对其做提取等操作会阻塞

julia> @fetchfrom 2 isready(c)       # 但在Worker 2中却是ready的
true

julia> @fetchfrom 2 take!(c)          # 而且能够顺利地提取出该元素
10
```

可见, 本地进程中的 Channel 对象 `c` 并没有被远端的 `put!()` 操作更新; 远端 Worker 更新的其实是其自有的拷贝, 所以也能够 `take!()` 出其中的元素; 但本地的 `c` 一直都没有 `ready`, 所以也会在调用 `fetch()` 或 `take!()` 时阻塞。

前文介绍的 `Future` 能够实现跨进程操作，可将远程调用的结果返回，实际是建立了远端 `Worker` 中计算结果的引用。通过 `Future` 的 `fetch()` 操作将远端对象调取的过程，可以理解为其中有个缓冲区大小为 1，能够容纳任意类型的 `Channel{Any}` (1) 结构，通过这样的结构将远端结果传递并缓存到本地进程中。

但是，这种基于 `Future` 的远程引用有诸多限制，例如，无法在任务运行期间随时传递数据。为此，`Julia` 提供了 `RemoteChannel` 类型，能够跨进程建立数据通道。该类型的内部结构为：

```
RemoteChannel{T<:AbstractChannel} <: Distributed.AbstractRemoteRef
  where::Int64
  whence::Int64
  id::Int64
```

可见，父类型与 `Future` 的父类型是一致的，而且成员字段也极为相似；区别在于该类型是参数化的，而且类型参数必须是 `AbstractChannel` 的子类型之一。

`RemoteChannel` 主要有以下构造方法，即：

```
RemoteChannel(pid::Integer=myid())
RemoteChannel(f::Function, pid::Integer=myid())
```

其中，`pid` 是某个 `Worker` 的 PID，默认是 `myid()` 即当前进程；函数 `f()` 由 `pid` 所指向的 `Worker` 执行，需返回 `AbstractChannel` 类型。

例如 `RemoteChannel(()->Channel{Int}(10), pid)`，会在远端 `Worker pid` 中创建一个类型为 `Int`，缓存大小为 10 的 `Channel` 对象，同时返回 `RemoteChannel` 对象到本地进程中，记录了该远端 `Channel` 对象的远程引用。

创建 `RemoteChannel` 的对象可以看作是某个 `Worker` 中 `Channel` 对象的句柄，同样可以对其进行 `put!()`、`take!()`、`fetch()`、`isready()` 和 `wait()` 等操作，只不过作用的实际对象是远端 `Worker` 中的那个 `Channel` 对象。而且任何进程（包括本地进程）均可通过该 `RemoteChannel` 对象进行各种操作，实现数据的发送与获取等功能。

以 `RemoteChannel` 在 `Worker 2` 中创建一个 3 元素的 `Channel` 对象为例，代码如下：

```
julia> using Distributed

julia> r = RemoteChannel(()->Channel{Int}(3), 2)      # 创建Worker 2一个Channel的远程引用r
RemoteChannel{Channel{Any}}(2, 1, 4)

julia> @fetchfrom 2 put!(r, 20)                       # 在Worker 2中对其放入数据
RemoteChannel{Channel{Any}}(2, 1, 4)

julia> isready(r)                                     # 本地进程虽未对r做过操作，但r变成了ready
true

julia> fetch(r)                                       # 而且在本地进程中能够fetch到Worker 2
20                                                    放入的数据

julia> @fetchfrom 3 isready(r)                       # 在另外的进程Worker 3中同样是有用的
```



```

true
julia> @fetchfrom 3 take!(r)           # 也能在Worker 3中提取到该数据
20

```

可见, 该 RemoteChannel 在本地 Worker 1、远端 Worker 2 及 Worker 3 实现了共享, 各进程对其的操作, 对其他 Worker 都是可见的。

如上所示, RemoteChannel 在多个进程中建立了数据通道, 使得它们能够随时相互地传递数据, 在多进程的环境中实现了 Channel 类似的功能。至此, 我们便可以在多进程的环境中实现交互式的应用。

例如, 可以对第 14.3.1 节脚本 demo_mconsumer.jl 中的代码进行改造, 将基于 Channel 与协程的并发消费修改为基于 RemoteChannel 与多进程的并发操作。在脚本 demo_rconsumer.jl 中所输入以下代码:

```

using Distributed

addprocs(4)                                # 增加4个远端Worker

jobs = RemoteChannel{()>Channel{Int}}(32);
results = RemoteChannel{()>Channel{Tuple}}(32))

@everywhere function do_work(jobs, results)    # 在所有Worker中定义消费者函数
    while true
        job_id = take!(jobs)                  # 用for循环从输入通道jobs中读取任务数据
        exec_time = rand()
        sleep(exec_time)                      # 用sleep模拟处理过程
        # 将结果放入输出通道results中
        put!(results, (job_id, exec_time, myid()))
    end
end

function make_jobs(n)                        # 生产者
    for i in 1:n
        put!(jobs, i)                        # 往任务通道jobs中放入数据
    end
end;

n = 12;

@async make_jobs(n);                        # 将生产者加入任务调度中

# 在远端Worker中异步地启动4个消费者, 实现并行消费
for p in workers()
    @async remote_do(do_work, p, jobs, results)
end

@elapsed while n > 0                        # 打印消费者执行后的结果
    # 从结果通道中提取数据
    job_id, exec_time, where = take!(results)
    println("$job_id finished in $(round(exec_time, digits=2)) seconds on
        worker $where")
    global n = n - 1
end

```

```
end
```

执行后, 得到如下结果:

```
1 finished in 0.4 seconds on worker 5
4 finished in 0.47 seconds on worker 2
6 finished in 0.12 seconds on worker 2
3 finished in 0.92 seconds on worker 4
2 finished in 0.93 seconds on worker 3
7 finished in 0.64 seconds on worker 2
10 finished in 0.01 seconds on worker 2
5 finished in 0.93 seconds on worker 5
9 finished in 0.88 seconds on worker 3
8 finished in 0.97 seconds on worker 4
11 finished in 0.79 seconds on worker 2
12 finished in 0.98 seconds on worker 5
```

可见, 该例顺利地通过 RemoteChannel 将任务分发到了可用的 4 个远端 Worker 中。虽然数据由本地 Task 提供, 但多个远端进程同时作为消费者并发地消耗了该任务数据。

如上所述, RemoteChannel 类似于 Channel, 均可在并行环境中进行数据交换。差别在于 RemoteChannel 是跨进程的。在资源隔离的并行进程中, 大量的消息传递与数据移动很容易成为并发的瓶颈, 所以应尽量避免中间数据的传递与转移操作。如果一定需要数据交互, 或者需要利用并发处理大规模的数据区, 可考虑使用下面将介绍的共享数组的方式。

14.5 共享数组

直接在并行计算层面提供对数据共享的支持, 是 Julia 语言的特色之一。Julia 中的共享数组是基于多维数组建立的一片共享内存。当 Worker 取得某个共享数组的结构时, 便可对同一内存区进行读写或更改操作, 而且不再需要额外的数据传递或复制操作。

共享数组的类型为 SharedArray, 位于官方维护的 SharedArrays 包中, 与 Array 一样也是 AbstractArray 的子类型。我们介绍 @distributed 时的那个例子说明了共享数组的基本用法。

该例试图基于宏 @distributed, 通过 for 循环修改元素的值, 但没有成功, 因为该数组在各个 Worker 中都有独立的一份。如果我们使用共享数组作为操作对象, 便能够实现预期的功能, 即:

```
julia> using SharedArrays, Distributed                                # 在使用SharedArray类型前, 需引入包
                                                                    SharedArrays

julia> a = SharedArray{Float64}(1, 10)                             # 可像普通数组那样构造一个共享数组
1×10 SharedArray{Float64,2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

julia> @distributed for i = 1:10
    a[i] = i
end
Task (queued) @0x000000000f972e10
```

```
julia> a
1×10 SharedArray{Float64,2}:
 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
```

可见，在 Worker 1 中定义的共享数组 `a` 中的元素，成功地被修改为对应的下标值。而且该例中都不需要关心是否对 `@distributed` 的结果进行 `fetch()` 操作，因为 `for` 循环在执行中对数组 `a` 的操作已经直接生效了。

共享数组有自己的构造方法，原型为：

```
SharedArray{T,N}(dims::NTuple; init=false, pids=Int[])
```

其中，`T` 是元类型参数；`N` 是维度；`dims` 是各维的阶数（一般 `N` 可省略）；类型 `Ntuple` 是 `Tuple` 类型的一种特殊别称。

键值参数 `pids` 用于指定该共享数组要对哪些 Worker 开放可见，若不指定则默认将数组结构映射到所有可用的远程 Worker 中。可以通过函数 `procs()` 获得所有被映射的 Worker PID 列表。例如上例中：

```
julia> procs(a)
4-element Array{Int64,1}:
 2
 3
 4
 5
```

```
julia> workers()
4-element Array{Int64,1}:
 2
 3
 4
 5
```

可见，该共享数组被映射到了所有可用的远程 Worker 中，即所有 Worker 都可以对其进行共享存取操作。

另外一个键值参数 `init` 的默认值 `false` 可被替换，但必须是函数对象，提供数组元素的初始化方法。该函数会被所有相关的 Worker 进程调用，完成共享数组中映射区的元素初始化，例如：

```
julia> @everywhere using SharedArrays

julia> S = SharedArray{Int,2}((3,4), init = S -> S[SharedArrays.localindices(S)] =
myid())
3×4 SharedArray{Int64,2}:    # 可见12个元素被平均分配给4个Worker
 2  3  4  5
 2  3  4  5
 2  3  4  5
```

其中，`SharedArrays.localindices()` 是一个辅助函数，能够根据进程映射情况均匀地为每个 Worker 分配不重叠的共享数组索引区域（采用线性索引的方式）。当然也可以选择其他方式或自定义的方式为每个进程分配索引范围。我们可以通过 `@spawnat` 的方式分别查看各 Worker 控制的索引范围，如下：

```
julia> using SharedArrays, Distributed

julia> fetch(@spawnat 2 SharedArrays.localindices(S))
1:3

julia> fetch(@spawnat 3 SharedArrays.localindices(S))
4:6

julia> fetch(@spawnat 4 SharedArrays.localindices(S))
7:9

julia> fetch(@spawnat 5 SharedArrays.localindices(S))
10:12
```

对于共享数组的初始化，上例仅作为示例。在实际的开发应用中，可以根据情况选择合适的方法。

共享数组在读取、索引、迭代、修改等方面都与常规数据没有太多差别，而且在被映射的任意远程 Worker 中效率都是一致的。例如：

```
julia> S[3,2] = 7
7

julia> @spawnat 3 S[2,3] = 10
Future{3, 1, 72, nothing}

julia> S
3×4 SharedArray{Int64,2}:
 2  3  4  5
 2  3 10  5
 2  7  4  5
```

或者：

```
julia> for i in S
    println(fetch(@spawn i, " at ", myid()))
end
(2, " at ", 2)      # Worker 2读取元素2
(2, " at ", 3)      # Worker 3读取元素2
(2, " at ", 4)
(3, " at ", 5)
(3, " at ", 2)
(7, " at ", 3)
(4, " at ", 4)
(10, " at ", 5)
(4, " at ", 2)
(5, " at ", 3)
(5, " at ", 4)
(5, " at ", 5)
```

正因为共享，SharedArray 在多个 Worker 中有着相同的访问权，在对数据进行并行处理时，很容易出现冲突。例如：

```
@sync begin
    for p in procs(S)      # 遍历可用的Worker列表
        @async begin
```



```

        remotecall_wait(fill!, p, S, p)    # 在Worker p中执行fill!()操作, 将S中所有元素置为p值
    end
end
end

```

因为并发执行的存在, 多个 Worker 的 `fill!()` 操作的先后顺序是无法预料的, 而该程序中的共享数组 `S` 只会保留最后一个 Worker 执行的结果, 所以在执行完之后 `S` 的内容并不可控。

显而易见, 在不跨多个机器的并行计算中, 共享数据使得并发操作大规模数据具有了极高的效率, 在很多时候极为便利。但是正如上例所示, 这是一把双刃剑, 在使用的过程中需要对共享区进行精准的控制, 避免出现不可预料的结果。

另外, 在 Julia 的第三方包中, 还有一个支持分布式数组的 `DArray` 库[⊖]可以选用。该包提供的 `DArray` 结构通过自动的索引分配, 能够整合多个机器的内存资源, 从而可以操纵超出单机可用内存的超大数组。有兴趣的读者可以参考其官方文档。

14.6 方法小结

上文介绍了多种不同的异步处理与远程调用的并行化方法, 包括 Task 协程、进程等方式。但每种方法都各有千秋, 为了方便在开发过程中进行选择, 本节进行了简要总结, 详见表 14-2 所示。

表 14-2 并行化方法汇总

宏或方法	用途说明	备注
<code>Task()</code> 或 <code>@task</code>	将函数或表达式封装为 Task 协程	返回 Task 对象
<code>schedule()</code>	将 Task 加入到调度队列中	返回 Task 对象
<code>@async</code>	将 Task 加入到调度队列中	返回 Task 对象
<code>remotecall()</code>	在指定的远端进程中进行远程调用	返回 Future 对象
<code>@spawnat</code>	同 <code>remotecall()</code> , 但可直接支持表达式	返回 Future 对象
<code>@spawn</code>	在任意的远端进程中进行远程调用	返回 Future 对象
<code>@everywhere</code>	在所有的远端进程中进行远程调用	无任何返回
<code>remote_do()</code>	在指定的远端进程中进行远程调用, 无返回机制	无任何返回
<code>@distributed</code>	对 for 循环自动进行远程调用, 并可进行归约	返回聚合结果或 Task 对象
<code>remotecall_fetch()</code>	在指定的远端进程中进行远端调用后, 阻塞等待	返回计算结果
<code>@fetchfrom</code>	同 <code>remotecall_fetch()</code> , 但可直接支持表达式	返回计算结果
<code>@fetch</code>	在任意的远端进程中进行远端调用后, 阻塞等待	返回计算结果
<code>@sync</code>	等待 <code>@spawn</code> 、 <code>@spawnat</code> 、 <code>@distributed</code> 和 <code>@async</code> 完成	返回计算结果

⊖ 见 <https://github.com/JuliaParallel/DistributedArrays.jl>

其中 Task 协程的使用有两个步骤：一是封装函数或表达式为 Task 对象；二是将其加入到调度队列中。在使用中，一般先以 Task 构造函数或 @task 宏进行封装，再通过 `schedule()` 函数对其进行调度；也可以 @async 在一次调用中实现这两个步骤。

Task 被调度系统启动运行后，会在阻塞操作时自动切换，从而提高队列中所有 Task 的整体效率，但不是提高单个 Task 的性能。这种执行是异步的，只能在本地，不能跨进程或跨机器。

多进程并行时，需 Julia 环境中已经启动了多个进程，可以 `julia -p <n>` 方式，或代码中在远程调用之前执行 `addprocs(n)`，增加 `n` 个进程，本地时一般不超出 CPU 的核数。

跨进程的远程调用同样是异步的，会在调用方立即返回。有多种选择方式：

- ☐ 处理过程只在单个进程中执行，还是在所有进程中执行。
- ☐ 在调用时是否需要指定 Worker 的 PID，还是不需要而自动选择。
- ☐ 远程计算的结果是否需要返回 Future 引用，或者无须返回。

为能更清晰地对比不同的远程调用方法，将五种常用方式的特性列举在表 14-3 中。开发者可以根据实际需求选择合适的方法。

表 14-3 远程调用方法对比

<code>remotecall()</code>	<code>@spawnat</code>	<code>@spawn</code>	<code>@everywhere</code>	<code>remote_do()</code>
单进程	单进程	单进程	所有进程	单进程
指定 PID	指定 PID	自动选择	自动选择	指定 PID
返回 Future	返回 Future	返回 Future	无返回	无返回

返回的 Future 对象包含了远程结果的引用，可通过 `fetch()` 进行提取，但该操作会在远端结束前阻塞。如果需要远程调用的同时立即获得远端结果，可以使用 `remotecall_fetch()`、`@fetchfrom` 或 `@fetch` 方式；但这种方式会将异步行为变成同步操作，所以在并行化环境中除非必要，否则尽量不用。

另外还有宏 `@distributed`，能够将 for 循环中的每次迭代置入某个远端执行，并可对各层循环的结果进行归约。期间，远程调用、进程选择、提取远程引用结果等过程都是自动的、隐式的，开发者不需要特别的关注。

如果 `@distributed` 没有归约操作，整个 for 循环仍会被异步执行，后续的代码会被立即执行。若是后续代码受到 for 循环过程的影响，则需要某个机制能够等待 `@distributed` 启动的所有异步行为全部完成。

宏 `@sync` 具有 `wait()` 类似的功能，但不需要显式的对象，适用性更广。该宏能够等待其后表达式或封闭 (enclosed) 的代码块中所有以 `@async`、`@spawn`、`@spawnat` 和 `@distributed` 方式启动的远程调用完成。例如：

```
@sync @distributed for var = range
    # body
end

# 后继语句
```

其中, @sync 可确保“后继语句”在 @distributed 隐式启动的所有远程调用完成后执行。再例如:

```
julia> using Distributed
julia> addprocs(3);
julia> @time for i = 1:5
           @spawn sleep(2)
       end
0.000766 seconds (674 allocations: 34.719 KiB)
```

虽然 for 中让远程 Worker 都“耗时”2 秒, 但实际整体 for 循环基本无耗时, 因为都是异步的。为了等待 for 循环整体完成, 可在 for 之前加上 @sync 宏, 如下:

```
julia> using Distributed

julia> @time @sync for i = 1:5
           @spawn sleep(2)
       end
2.008066 seconds (752 allocations: 28.984 KiB)
```

可见, for 循环整体耗时基本是 Worker 的实际耗时。

除了上述介绍的各种并行方法外, Julia 可以在启动时通过 --machinefile 参数提供一份机器名单, 将多机器构造为一个集群。此时, 系统会通过 SSH 方式无密码地登录远程特定的机器并启动 Julia 工作进程。而且, Julia 提供了专门的集群管理器 (ClusterManager), 能够为多进程并行化提供进程启动、管理和网络通信等方面的支持。

该集群管理器有 Distributed.LocalManager 与 Distributed.SSHManager 两个子类型。前者用于在同一个服务器中启动更多的工作进程, 从而可以充分利用本地硬件的多核优势; 后者则通过 SSH 在远程服务器中启动工作进程。与此有关的内容, 本书不做介绍, 感兴趣的读者可参见官方文档。

另外, 自 Julia v0.5 版本后, 原生提供了多线程 (multi-threading) 机制。但该功能仍处于试验阶段, 未来可能会发生变更, 所以本书也不作介绍, 有兴趣的读者可以参考相关资料。

混合编程

Julia 是一门新兴的语言，正在蓬勃迅速地发展。在我们将一些工作迁移到该语言的过程中，必然会涉及既有成果的继承问题。Julia 在设计时提供了混合编程的机制，能够与成熟的语言进行融合，调用 C/C++、Python、Java 等库，或者被这些语言调用，甚至可以相互嵌入代码。这样，我们在学习使用 Julia 语言时，便能够借力这些普及语言的生态环境，实现跨语言的集成，从而能快速地转移到新的语言平台。

建议在学习本章的内容时，能够在 Linux 或 MacOS 平台上进行。

15.1 运行外部程序

在 Julia 的代码中，可以调用系统已经安装的外部程序。方式很简单，将命令调用的执行语句以反引号 `` 标识出来，并创建一个 Cmd 类型的对象，然后在需要的位置以该对象为参数调用 run() 函数。此后，该命令语句便会执行。这种方式与 Shell 中执行外部命令的语法是相似的。例如：

```
julia> cmd = `echo hello`  
`echo hello`  
  
julia> typeof(cmd)  
Cmd  
  
julia> run(cmd)  
hello  
Process(`echo hello`, ProcessExited(0))
```

但是这种运行方式不能获得程序的输出，仅会将执行中的结果输出到 stdout 流中。如果需要获得外部程序执行的结果，可使用 read() 函数，例如：


```
julia> outs = read(pipeline(stdin, cmd))
6-element Array{UInt8,1}:
 0x68
 0x65
 0x6c
 0x6c
 0x6f
 0x0a

julia> String(outs)
"hello\n"
```

可见，Julia 在运行外部命令时，不需要 Shell 或系统命令行环境的支持。在执行时，Julia 会直接解析命令的语法，并进行变量展开或语句分离等操作，然后在内部环境中启动一个子进程，并使用 fork 或 exec 启动该外部程序。

15.2 调用 C/C++

C 语言历史悠久，已经广泛地应用于各种领域，尤其是在资源控制、性能敏感的场景中，积累了大量高性能、成熟的库。Julia 在与 C 语言混合编程方面，不需要任何额外的第三方粘合语言，便可直接调用与嵌入 C 语言，而且不会编译生成中间代码，所以能够充分地利用 C 语言的各种库。

15.2.1 链接库操作

通过 C 语言链接库使用既有功能是基本的方式。Julia 在 Base.Libdl 模块中提供了操作 C 语言动态链接库的各种函数。

例如，加载一个动态库可使用 Libdl.dlopen() 函数，其原型为：

```
dlopen(libfile::AbstractString [, flags::Integer])
```

或者：

```
dlopen_e(libfile::AbstractString [, flags::Integer])
```

调用成功后会返回链接库的句柄，两种方式的区别仅在于发生异常时是返回 NULL 还是直接报错。其中，参数 libfile 为链接库的名称或者完整路径，如果未给全路径，需要在系统目录或系统变量 DL_LOAD_PATH 列出的目录中找到库。库的扩展名 (.so/.dll/.dylib 等) 可省略，可以通过内置常量 Libdl.dlext 获知。可选参数 flags 是指定链接库加载的模式，在不同的系统中有所不同，在 Linux 下各种可用模式意义如表 15-1 所示。

表 15-1 动态链接库加载模式

模 式	意 义
RTLD_LAZY	在 dlopen 返回前，对于动态库中的未定义的符号不执行解析（只对函数引用有效，对于变量引用总是立即解析）

(续)

模 式	意 义
RTLD_NOW	在 dlopen 返回前, 解析出所有未定义符号, 如果解析不出来, 在 dlopen 会返回 NULL, 错误为: : undefined symbol:
RTLD_GLOBAL	动态库中定义的符号可被其后打开的其他库解析
RTLD_LOCAL	与 RTLD_GLOBAL 作用相反, 动态库中定义的符号不能被其后打开的其他库重定位。如果没有指明是 RTLD_GLOBAL 还是 RTLD_LOCAL, 则默认为 RTLD_LOCAL
RTLD_NODELETE	在 dlclose() 期间不卸载库, 并且在以后使用 dlopen() 重新加载库时不初始化库中的静态变量。这个 flag 不是 POSIX-2001 标准
RTLD_NOLOAD	不加载库。可用于测试库是否已加载 (dlopen() 返回 NULL 说明未加载, 否则说明已加载), 也可用于改变已加载库的 flag。例如, 先前加载库的 flag 为 RTLD_LOCAL, 用 dlopen(RTLD_NOLOAD RTLD_GLOBAL) 后 flag 将变成 RTLD_GLOBAL。这个 flag 不是 POSIX-2001 标准
RTLD_DEEPBIND	在搜索全局符号前先搜索库内的符号, 避免同名符号冲突。这个 flag 不是 POSIX-2001 标准

这些模式在 dlopen() 时会转换到 POSIX 标准中 dlopen 命令的对应参数。但如果不指定 flags 参数, 则会取默认值: 在 MacOS 中为 RTLD_LAZY | RTLD_DEEPBIND | RTLD_GLOBAL, 其他系统则为 RTLD_LAZY | RTLD_DEEPBIND | RTLD_LOCAL。

flags 参数的主要作用在于, 连接器何时将库引用到导出的符号中, 或绑定时放在进程的局部域还是全局域。例如, RTLD_LAZY | RTLD_DEEPBIND | RTLD_LOCAL 允许链接库符号可以在其他链接库可见并可用, 能解决链接库之间的依赖关系。

如果在使用 dlopen() 时, 仅知道链接库的名称而不无法提供完整的路径, 则可以使用 Julia 的内置函数 Libdl.find_library(libnames, locations), 依序在 locations 给定的目录中及内部数组常量 DL_LOAD_PATH 列出的目录表中, 搜索名为 libnames 的动态链接库的完整路径, 然后便可将搜索到的路径作为参数传入 dlopen() 函数, 加载该库文件以获得其句柄。

基于库的句柄, 可通过 dlsym() 函数提取其中函数的指针, 然后便可使用该函数了。与函数 dlopen() 类似, dlsym() 函数也有两种方法, 如下:

```
dlsym(handle, func)
```

或者:

```
dlsym_e(handle, func)
```

其中, 参数 handle 是 dlopen() 成功后返回的链接库句柄; func 是 Symbol 类型的函数名。关于如何使用获得的名为 func 的函数指针, 会在后文详细介绍。

在使用完库中的功能后, 不要忘记调用 dlclose() 函数关闭加载的链接库资源。

15.2.2 函数调用

通过调用 dlopen() 以及 dlsym() 函数后, 我们获得了 C 语言链接库中函数 func

的指针。然后便可通过 `ccall()` 函数执行 `func` 的功能了。`ccall()` 函数的原型为：

```
ccall(func_pointer, ReturnType, (ArgType1, ...), ArgValue1, ...)
```

其中, `func_pointer` 是 `dlsym()` 函数返回的函数指针; 参数 `ReturnType` 是 Julia 类型系统中支持的类型名称, 用于指定该函数执行时的返回值类型; 元组参数 `(ArgType1, ...)` 用于说明该 C 函数所需各参数的类型; 后续的 `ArgValue1, ...` 等则是与类型元组对应的实参值。

另外, `ccall()` 函数还有一个方法, 可以直接加载库中的函数, 原型为:

```
ccall((func, library), ReturnType, (ArgType1, ...), ArgValue1, ...)
```

其中, 字符串 `library` 参数是动态链接库的路径; 参数 `func` 需以 `Symbol` 类型提供需要提取的 C 函数名称。

`ccall()` 函数有些特殊的要求: `(func, library)` 参数只能是字面常量; 同时, 说明类型的元组参数 `(ArgType1, ...)` 也只能在代码中直接写明, 不能使用变量方式提供。

需特别注意的是, 类型元组只有一个元素时, 后面必须紧跟一个逗号作为结尾。例如, 函数 `func` 只需要一个整型参数, 在提供类型说明时不能写为 `(Int32)` 而应写为 `(Int32,)`, 这是因为前者会出现语法歧义 (参见 6.9.1 节)。

由于跨语言, 不同语言在类型方面会存在很多差异, 所以 Julia 内部声明了专门的类型与函数, 用于 C 类型和 Julia 类型之间的兼容转换。表 15-2 给出了常见的类型对照, 更多内容请参阅相关资料。

表 15-2 Julia 类型与 C 类型对比

C 类型	Julia 兼容类型	Julia 基准类型	C 类型	Julia 兼容类型	Julia 基准类型
unsigned char	Cuchar	UInt8	float	Cfloat	Float32
bool(仅 C++)	Cuchar	UInt8	double	Cdouble	Float64
short	Cshort	Int16	size_t	Csize_t	UInt
unsigned short	Cushort	UInt16	void		Nothing
int, BOOL(C 中常见别称)	Cint	Int32	void*		Ptr{Nothing}
unsigned int	Cuint	UInt32	T*(T 为自定义结构类型)		Ref{T}
long long	Clonglong	Int64	char*(或 char[], 即字符串)		NUL(0) 结尾时为 Cstring, 否则为 Ptr{UInt8}
unsigned long long	Culonglong	UInt64	char** 或 *char[]		Ptr{Ptr{UInt8}}

表 15-2 中有两个特别的类型——`Ptr{T}` 和 `Ref{T}`, 这里说明一下。

在 6.8.3 节中介绍过, `Ptr{T}` 是参数化元类型, 表达一个指向类型 `T` 内容的内存地址, 对应于 C 语言中的 `T*` 类型; `Ref{T}` 对象则指代 `T` 类型数据区的引用, 对应于 C/C++ 中

的引用 `T&` 类型。类似地, Julia 不保证该 `Ptr{T}` 的内存区可用; 而 `Ref{T}` 对应的引用由 Julia 分配管理并在其作用期提供持续保护, 可保证有效可用, 所以不存在无效 (`NULL`) 的 `Ref`。当 `Ref` 作为 `ccall` 的参数时, `Ref` 对象会转换到指向其引用数据区的原生指针。

调用 `ccall()` 使用元类型 `T`、`Ref{T}` 及 `Ptr{T}` 时, 有些常规约定: 普通的非指针数据因为是传值, 所以应声明为 `T` 类对象; `Ref{T}` 可用于被 C 函数内部改变的指针参数, 但需保证是有效的数据; `Ptr{T}` 则用于指定返回数据的指针, 并表明该内存仅被 C 内部管理。

关于调用 C 时更多的转换规则与用法, 限于篇幅不作过多介绍。下面我们通过实际的例子了解 `ccall()` 等的用法。例如, 可通过库 `libc` 调用函数 `clock()` 取得系统时间戳:

```
julia> t = ccall(:clock, "libc", Int32, ())
2292761

julia> typeof(t)
Int32
```

其中, 第二个参数 `Int32` 指定了返回值的类型; 第三个参数用于传入函数所需的参数值, 因为该函数无须传参, 所以是空的元组对象。

再例如, 调用 C 函数 `getenv()` 获得环境变量字符串的指针:

```
julia> path = ccall(:getenv, "libc", Cstring, (Cstring,), "SHELL")
Cstring(@0x00007fff5fbffc45)
```

其中, 返回值指定为兼容类型 `Cstring`, `getenv()` 函数的参数类型同样如此; 之后传入字符串参数 “SHELL” 要求获知当前的 Shell 类型。执行后, 获得 `Cstring` 类型的结果, 但还需要做一下转换, 即:

```
julia> unsafe_string(path)
"/bin/bash"
```

其中, 函数 `unsafe_string()` 用于将 `Cstring` 对象转为 Julia 的字符串。

在实践中, 可以用 Julia 函数将对 C 函数的调用进行封装: 一是可以将 C 库中的各种变化进行隔离; 二是有利于功能复用。仍以环境变量为例:

```
function getenv(var::AbstractString)
    val = ccall(:getenv, "libc",
                Cstring, (Cstring,), var)
    if val == C_NULL
        error("getenv: undefined variable: ", var)
    end
    unsafe_string(val)
end
```

然后就可以像普通 Julia 函数那样使用:

```
julia> getenv("SHELL")
"/bin/bash"

julia> getenv("FOOBAR")
getenv: undefined variable: FOOBAR
```

有时候, 我们会面临另外一种情况——在调用 C 函数时, 该函数接收的参数不是普通

的类型，而是函数指针。此时可通过 Julia 的 `cfunction()` 函数包装成 C 函数可接受的形式，其原型为：

```
cfunction(func::Function, ReturnType::Type, ArgTypes::Type)
```

其中，`func` 是 Julia 中定义的函数名；`ReturnType` 是返回值的类型；`ArgTypes` 是 Tuple 结构的类型参数，告知各函数的参数类型。

假设有个 `Float64` 元素的一维数组 `A`，我们希望能不使用 Julia 内置的排序函数，而是调用标准 C 库中的经典函数 `qsort()`，原型如下：

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compare)(const void *a, const void *b));
```

其中，参数 `base` 是元素个数为 `nmemb` 的数组指针；每个元素的大小为 `size` 字节；`compare` 是一个回调函数指针，提供比较方法并返回整型值标识两者的大小关系。

首先，用 Julia 语言定义一个比较函数：

```
julia> function mycompare(a::T, b::T) where T
           return convert{Cint, a < b ? -1 : a > b ? +1 : 0}::Cint
       end
mycompare (generic function with 1 method)
```

这是一个参数化函数，实现对 `T` 类型的两个变量的比较。注意其中的类型 `Cint`，虽然定义的是 Julia 函数，但仍需使用兼容类型。

然后，我们就需要使用 Julia 的 `@cfunction()` 对该函数进行包装，如下：

```
julia> const mycompare_c = @cfunction(mycompare, Cint,
                                     (Ref{Cdouble}, Ref{Cdouble}))
Ptr{Nothing} @0x000000001f458500
```

最后，便可以在 Julia 中执行 `qsort` 的调用过程了，即：

```
julia> A = [1.3, -2.7, 4.4, 3.1]
4-element Array{Float64,1}:
 1.3
-2.7
 4.4
 3.1

julia> ccall(:qsort, Nothing, (Ptr{Cdouble}, Csize_t, Csize_t, Ptr{Nothing}),
            A, length(A), sizeof(eltype(A)), mycompare_c)

julia> A
4-element Array{Float64,1}:
-2.7
 1.3
 3.1
 4.4
```

可见，Julia 中的 `A` 数组在执行 `ccall()` 之后，被成功排序并就地修改为有序的序列。

15.2.3 数据访问

除了能够调用 C 库中的函数，也可以通过 `cglobal()` 函数直接访问其中的全局变量。

该函数的原型为：

```
cglobal((name, library) [, type=Nothing])
```

类似于 `ccall`，参数 `name` 是该变量的 `Symbol` 类型名，`library` 是库名字符串，`type` 告知该全局变量的类型。

该函数被调用后，会返回 `library` 中名为 `name` 的全局变量的指针，类型为 `Ptr{type}`；如果不指定 `type`，则默认为 `Ptr{Nothing}` 类型。

例如，要取得 `libc` 库中的全局变量 `errno` 的值，可以先调用 `cglobal()` 获取其指针，如下：

```
julia> cglobal(:(errno), :libc, Int32)
Ptr{Int32} @0x00007f418d0816b8
```

其中，`Ptr{Int32}` 类型的值是该变量的指针值。

此后便可调用 `unsafe_load()` 函数取得其在 Julia 类型中的值，或者使用 `unsafe_store!()` 函数修改 `errno` 在 C 库 `libc` 中的值。

之所以对 C 库中内容的读写函数均有 `unsafe` 标识，这是因为如果提供的参数是无效的指针或类型声明，会导致 Julia 环境本身陡然崩溃。下面详细介绍这方面的机制。

对 C 库指针进行读操作的函数原型为 `unsafe_load(ptr[, index])`，其中 `ptr` 是 `cglobal()` 等操作返回的 `Ptr{T}` 类型的对象，指向 C 库中的某个变量。该函数会将偏移在 `index` 处的内容复制到某个 Julia 对象中，而且自动处理了 Julia 的 1-based 的索引方式，所以访问过程类似于 `ptr[index]` 这种方式。其返回时会创建新的 Julia 对象，并将 `ptr` 指向的内容复制到该对象中，此后内存中的原内容便可被安全地释放或清理掉。

但是，如果 `T` 被指定为 `Any` 类型，则认为 `ptr` 指向的内容是某个真实 Julia 对象的引用，即该内容不是实际的值，而仍是指针（即 `j1_value_t*`）；当 `unsafe_load` 时，返回的结果仍是引用，不会复制真实的对象，此处需非常小心，因为此时该函数实际上创建了新的引用。如果 GC 无法得知该次新引用，则无法确保真实对象不会被过早地释放。例如，对象并非是由 Julia 分配管理的，那新引用将永远不会被 Julia 的 GC 处理（C/C++ 是没有 GC 的）。

为此，可以通过 `unsafe_pointer_to_objref(ptr)` 调用将 `ptr` 转为 Julia 中的引用，当然也可以调用 `pointer_from_objref(v)` 将 Julia 中的值 `v` 转为 `j1_value_t*` 指针（与 `Ptr{Nothing}` 对应）。

若是以 Julia 对象修改库中的值，则写函数 `unsafe_store!(ptr, value[, index])`，但目前仅支持元类型或其他无指针内容（`Pointer-free`, `isbits`）的不可变 `struct` 类型。

如果指针是普通数据（元类型或不可变 `struct` 类型）的数组，函数 `unsafe_wrap(array, ptr, dims[, own=false])` 会更为有用。通过设置 `own` 参数为 `true` 值，Julia 便可接管内存的管理权，此后当对象 `array` 用完，便可调用 `free(ptr)` 释放该对象。但如果 `own` 设为 `false` 值，则调用者必须确保在所有访问完成前内存区仍是有效的。



注意 在 Julia 中对 `Ptr` 类型做算术计算时, 比如累加一个整型数值 `v`, 则 `Ptr` 对象会移动 `v` 个字节数; 但对 C 指针累加 `v` 值时, 则是向后移动 `v` 个元素。这样的区别, 需在 Julia 与 C 混合编程时注意。

15.2.4 C++ 接口

在与 C++ 混合编程方面, 有两个包提供了强大的支持——`Cxx`[⊖]包与 `CxxWrap`[⊖]包。前者能够让开发者在 Julia 中直接嵌入 C++ 代码, 例如:

```
julia> jnum = 10;

julia> cxx"""
    void printme(int x) {
        std::cout << x << std::endl;
    }
    """

julia> @cxx printme(jnum)
10
```

而后者 `CxxWrap` 则支持相互独立编程, 再通过封装机制进行语言的交互, 有兴趣的读者可通过其官方文档做详细的了解。

15.3 嵌入 C/C++

本节介绍如何在 C/C++ 中调用 Julia 代码, 需要先熟悉 C/C++ 语言。如果不熟悉, 可将此部分作为选读内容, 或者尝试学习一下这个强大而有生命力的语言。

为了能够被其他语言嵌入做混合开发, Julia 提供了专门的 C 语言版 API。而且该 API 不仅仅用于 C/C++ 语言, 还可以作为嵌入 Python、C# 等语言的中间桥梁。

当然, 在使用中需要配置环境变量和相关编译运行环境。例如, 要设定环境变量 `JULIA_DIR` 指向 Julia 的安装目录, 设定 `Sys.BINDIR` 指向 `$JULIA_DIR/bin` 目录, 更多的内容可参见相关资料。

先举个 Hello World 式的简单例子。假设有个 C 语言文件 `test.c`, 其内容为:

```
#include <julia.h>      # Julia的API提供的头文件

int main(int argc, char *argv[])
{
    /* 必需: 初始化Julia上下文 */
    jl_init();

    /* 运行某个Julia命令 */
    jl_eval_string("print(sqrt(2.0))");
}
```

⊖ 见 <https://github.com/Keno/Cxx.jl>

⊖ 见 <https://github.com/JuliaInterop/CxxWrap.jl>


```

/* 强烈推荐: 通知Julia该程序要终止了, 其可以清理未完成的写操作并运行相关终止过程 */
jl_atexit_hook(0);
return 0;
}

```

如果你熟悉 C 语言, 能够看出这个基本结构非常简单, 包括了初始化与清理过程。

其中的初始化函数 `jl_init()` 是必须要调用的, 会根据环境变量尝试自动确定 Julia 的安装目录, 如果要加载特定的 Julia 镜像, 则可使用 `jl_init_with_image()` 函数; 另外, 在程序结束时, 强烈推荐调用 `jl_atexit_hook()` 函数, 以便 Julia 内部进行清理工作。

此后, 便可在 Linux 或 MacOS 下使用 GCC 对该源文件进行编译, 大致的命令如下:

```
gcc -o test -fPIC -I$JULIA_DIR/include/julia -L$JULIA_DIR/lib test.c -ljulia
```

其中, `-ljulia` 是要链接 Julia 的库文件。

实践中可能还需要其他相关的库, 例如 `libstdc++` 等, 可根据实际情况增加。为了避免繁杂的链接编译问题, Julia 提供了专门的工具 `julia-config.jl`, 用于生成当前系统环境下需要的编译参数, 而且能够用于 Makefile。例如:

```

JL_SHARE = $(shell julia -e 'print(joinpath(Sys.BINDIR, Base.DATAROOTDIR, "julia"))')
CFLAGS   += $(shell $(JL_SHARE)/julia-config.jl --cflags)
CXXFLAGS += $(shell $(JL_SHARE)/julia-config.jl --cflags)
LDFLAGS  += $(shell $(JL_SHARE)/julia-config.jl --ldflags)
LDLIBS   += $(shell $(JL_SHARE)/julia-config.jl --ldlibs)

```

对于熟悉 C/C++ 的读者, 这些都是编译器常规的参数。例子中的 `shell` 需要替换为系统可用的真实 Shell 程序; 变量 `Sys.BINDIR` 需要在系统环境变量中进行有效的配置, 指向真实的 Julia 安装目录; `Base.DATAROOTDIR` 是 Julia 中的常量, 指向安装目录中的 `share` 文件夹。



提示 GNU make 是 Linux 类操作系统极为常用的编译管理工具, 只要基于特定的语法规则编写 makefile 文件, 即可实现复杂的代码管理、编译与实施。还有另外一个更为便捷、强大的编译工具 CMake, 有兴趣的读者可以尝试, 此处限于篇幅不再赘述。

另外, 类似于 Julia 调用 C 函数, 在 C/C++ 语言调用 Julia 时也需要处理类型转换问题。在 API 中, 前缀 `jl_box_` 系列函数用于将 C 值转换到 Julia 类型; 前缀 `jl_unbox_` 系列函数用于将 Julia 类型转为 C 类型。例如:

```

jl_value_t *a = jl_box_float64(3.0);
jl_value_t *b = jl_box_float32(3.0f);
jl_value_t *c = jl_box_int32(3);

```

其中, `jl_value_t*` 为指向堆中分配的 Julia 对象指针。可以这样理解: `box` 是打包, 而 `unbox` 是解包, 对应着语言中类型转换的不同方向。

对上面的例子稍加改造, 便可演示 `unbox` 的使用方法, 如下:

```

#include <julia.h>

int main(int argc, char *argv[])

```



```

{
    jl_init();
    jl_value_t *ret = jl_eval_string("sqrt(2.0)");

    if (jl_typeis(ret, jl_float64_type)) {

        double ret_unboxed = jl_unbox_float64(ret);
        printf("sqrt(2.0) in C: %e \n", ret_unboxed);
    }
    else {
        printf("ERROR: unexpected return type from sqrt(::Float64)\n");
    }

    jl_atexit_hook(0);
    return 0;
}

```

其中, `jl_unbox_float64` 将 `jl_value_t*` 指向的 Julia 类型的值解包出来, 转换到原生的 C 类型 `double`; 用到的 `jl_typeis()` 函数可以测试一个值是否是特定的 Julia 类型。另外, 具有 `jl_isa()`, `jl_is_` 前缀的系列函数有着类似的功能。

上例中, 虽然 `jl_eval_string()` 函数可以获得 Julia 表达式的结果, 但无法传参。所以 Julia 的 API 提供了另外一个函数 `jl_call()`, 其用法可简单示例如下:

```

jl_function_t *func = jl_get_function(jl_base_module, "sqrt");
jl_value_t *argument = jl_box_float64(2.0);
jl_value_t *ret = jl_call1(func, argument);

```

其中, `jl_base_module` 是内置的常量, 用于指代 Julia 的 Base 模块。使用 `jl_get_function()` 将其中 `sqrt()` 函数提取为 C 语言可以操作的类型 `jl_function_t*`, 然后通过 `jl_box_float64()` 将 C 语言的值转为 Julia 可用的类型值 `jl_value_t*` 以作为参数; 再将取得的函数指针 `func` 和 `argument` 传入 `jl_call1()` 函数调用, 并将计算的值返回到 `jl_value_t*` 中。

注意, 其中的函数 `jl_call` 带有后缀 1, 这是因为该函数支持多种参数形式, 还有 `jl_call0()`、`jl_call2()` 和 `jl_call3()` 等原型可用。因为 C 语言不支持多态, 所以需要不同函数名标识。这几个函数主要区别在于参数个数不同, 如果需要传入任意的参数, 则可以考虑如下的原型:

```

jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t nargs)

```

其中, 第 2 个参数 `args` 是 `jl_value_t*` 类型的数组, 用于接收一系列参数值; `nargs` 用于说明 `args` 中记录的参数个数。

另外, 数组结构本身作为参数也较常见。在 Julia 的 API 中, 数组结构的类型为 `jl_array_t*`, 能够被 Julia 和 C 共享, 而无须以复制的方式传递。该类型是一个复合结构, 不但存储了数据序列的地址指针, 同时还记录了数组的大小信息。但因为 C 是强类型语言, 需要先声明才能使用一个变量, 所以使用 API 创建数组需要两个步骤。

例如, 要定义一个一维数组, 基本用法为:

```
j1_value_t* array_type = j1_apply_array_type(j1_float64_type, 1);
j1_array_t* array_data = j1_alloc_array_1d(array_type, 10);
```

首先通过 `j1_apply_array_type()` 声明 `double` 类型的数组, 然后 `j1_alloc_array_1d()` 函数会在 `array_type` 指向的内存中分配 10 个元素的空间, 并返回 `j1_array_t*` 类型的变量 `array_data`。

或者, 可以封装在 C 语言中已经存在的数组数据, 例如:

```
double *existingArray = (double*)malloc(sizeof(double)*10);
j1_array_t *x = j1_ptr_to_array_1d(array_type, existingArray, 10, 0);
```

其中, 最后一个参数 0 相当于布尔值 (C 语言中没有专门的布尔型), 可指定封装的数组内存区的管理权是否转移到 Julia 中。

无论采用上述两种方式中哪一种, 在得到 `j1_array_t*` 类型的变量后, 如果要基于 C 语言的方式对其进行操作, 便需要将其转换到 C 类型, 如下:

```
double *xData = (double*)j1_array_data(x);
```

其中, 函数 `j1_array_data()` 用于解包工作。需要注意的是, 得到的 `double*` 类型的 `xData` 变量和 `j1_array_t*` 类型的变量 `x` 指向了同一块内存区, Julia 和 C 已经对这片内存实现了共享, 所以两者可以同时对该数组结构进行操作。例如, 在 C 语言中:

```
for(size_t i=0; i<j1_array_len(x); i++)
    xData[i] = i;
```

或通过 Julia 内的函数对其进行操作, 例如:

```
j1_function_t *func = j1_get_function(j1_base_module, "reverse!");
j1_call1(func, (j1_value_t*)x);
```

其中, `reverse!()` 是 Julia 中的函数, 会原地改变该数组数据。

当然, 此例中也可使用 `reverse()` 函数实现反转序列, 但需从返回值取得新结果:

```
j1_function_t *func = j1_get_function(j1_base_module, "reverse");
j1_array_t *y = (j1_array_t*)j1_call1(func, (j1_value_t*)x);
```

然后通过 `j1_call1()` 函数将结果转换即可继续后面的操作。

如果操作二维数组, 过程是极为类似的, 例如:

```
// 创建Float64类型的2维数组, 并分配内存
j1_value_t *array_type = j1_apply_array_type(j1_float64_type, 2);
j1_array_t *x = j1_alloc_array_2d(array_type, 10, 5);
```

```
// 取得指针
double *p = (double*)j1_array_data(x);
// 获得维数
int ndims = j1_array_ndims(x);
// 获得每一维的大小
size_t size0 = j1_array_dim(x, 0);
size_t size1 = j1_array_dim(x, 1);
```

```
// 更改数据
for(size_t i=0; i<size1; i++)
```

```
for(size_t j=0; j<size0; j++)
    p[j + size0*i] = i + j;
```



注意 C 语言中索引是从 0 开始的，而 Julia 是基于 1 进行索引的。所以在混合编程时，如果要操作数组，需要注意这两个语言在这方面的差异。

15.4 与 Python 互调

通过 JuliaPy 项目下的 PyCall.jl^①包可以在 Julia 中调用 Python；相对地，可以通过 pyjulia.jl^②包在 Python 中实现对 Julia 的调用。本节仅对 PyCall 进行简单的介绍，更多内容请参考其文档。

在通过 Pkg.add("PyCall") 安装完 PyCall 之后，可像写 Python 程序那样，在 Julia 中编写代码，几乎是无缝的对接，还可以在 Julia 的 REPL 中直接使用。PyCall 内部会自动地对各种类型进行转换，包括数值、布尔型、IO 流、日期/周期、元组、数组、字典和函数等。

以一个简单的例子演示 PyCall 包的用法，例如：

```
julia> using PyCall;
julia> @pyimport math;
julia> math.sin(math.pi / 4) - sin(pi / 4)
0.0
```

其中，使用 @pyimport 宏导入 Python 的 math 模块后，便能直接通过点操作符使用其中的成员函数 sin() 以及 Python 中的常量 pi；在通过 Python 计算 math.sin(math.pi/4) 时，同时可使用 Julia 中的函数计算 sin(pi/4)。两者的计算结果是相同的。

但在导入 Python 的子模块时，需要在使用成员操作符指定子模块的同时定义一个别名才能在 Julia 正常使用，这是因为点操作符会与 Julia 中的语法产生歧义。所以，应如下例所示那样，使用 Python 的子模块：

```
julia> @pyimport numpy.random as nr;
julia> nr.rand(3,4)
3×4 Array{Float64,2}:
 0.54323  0.113636  0.6938   0.556447
 0.145428 0.0893743 0.510443 0.0909521
 0.152075 0.422302  0.199568 0.454841
```

其中，调用 Python 的随机函数生成了 Julia 的二维数组。

在 PyCall 中，Python 与 Julia 的高维数组转换接口是基于 Numpy 数组结构的。默认情况下从 Julia 到 Python 时不需要拷贝，但从 Python 到 Julia 会进行拷贝处理。如果要避免这种额外的拷贝操作，可借助 PyArray 类型。

① 见 <https://github.com/JuliaPy/PyCall.jl>

② 见 <https://github.com/JuliaPy/pyjulia.jl>

另外, Julia 中的函数对象还可以作为 Python 函数的参数传入。例如, 基于 Python 的 SciPy 模块求解一个一元多项式的根, 如下:

```
julia> @pyimport scipy.optimize as so;
julia> so.newton(x -> cos(x) - x, 1)
0.7390851332151607
```

其中, `newton()` 是 Python 的函数, 其参数 `x -> cos(x) - x` 是 Julia 中定义的匿名函数。

关于 Julia 与 Python 混合编程的更多内容, 本书不再赘述, 有兴趣的读者可参阅官方文档资料。

Julia 编程规范

在掌握一门语言的基本语法后，就可以一试身手，编写一些程序了。但实际上，一切才刚刚开始！

在计算资源的各种限制下，在运行指标的各种要求中，若要使用某种语言编写出足够可靠、高效、稳定的程序，还需对该语言的细节有更深入的了解。掌握该语言的优缺点，经过大量的练习与实践，我们才能驾轻就熟，真正控制着某个语言按照设定的预期目标，实现想要的程序系统。

为了能够帮助读者在 Julia 的学习之路上更进一步，少走些弯路，尽快提高编程技巧，特专辟本章，对 Julia 语言相关的规范、原则、各种注意事项及语言设计者给出的多方建议进行详尽的介绍。

本章首先介绍编程中不可忽视的重要方面——Julia 文档注释系统；然后在如何编写出高性能 Julia 程序方面，给出各种注意事项，帮助学习者规避一些问题；还会比较 Julia 语言与 Python、Matlab 及 R 语言的不同之处，也是对其特点的总结；最后在代码风格方面给出建议。

虽然本章可作为选读内容，但从笔者多年的编程经验来看，深刻理解本章内容，能给今后的开发工作带来很大的帮助。

16.1 文档注释

代码注释经常会被忽略，尤其是初级程序开发人员，但事实上其重要作用往往超过程序的设计文档。

无论对代码多么熟悉，开发时又是多么花心思，通常时隔一个月甚至两周后，开发者

一般就会忘记当时的实现逻辑，想不出当时为何这么实现。所以，足够详实的代码注释，尤其关键位置处的重要标识，会给后期的功能优化、修正维护带来极大的裨益。可以说，代码注释的重要性，无论怎么去阐述都不为过。

Julia 自 v0.4 版本之后，便提供了内置的文档注释系统，可以很方便地对函数、类型、对象及语句进行描述，直接支持 Markdown 格式^①，能够生成非常实用、方便的说明文档。因篇幅所限，Markdown 语法格式的详细介绍需要读者自行查阅，下文仅对 Julia 别具特色的文档系统进行阐述。

最简单的注释是“一句话就可说清楚”的单行注释方式，以井号 # 作为注释内容的前缀，一般放于语句的上一行或当前行尾部。这种注释方式在前文大量的示例代码中已经广泛使用，此处不再赘述。

对于多行注释，一般在注释内容的上下行使用 `#=` 与 `=#` 两个标记符进行界定，例如：

```
#=
    这是一个多行注释，
    这是多行注释的另外一行
=#
function some()
    # 单行注释
end
```

更为强大的注释方式是 Julia 提供的文档字符串（docstrings）。

在 Julia 的脚本代码中，类型声明、函数与宏定义等上一行中单独出现的字符串对象都被认为是注释内容。例如：

```
"这是对函数foo的注释"
foo(xs::Array) = ...
```

其中，`foo()` 之上的单行字符串被编译器处理为该函数的注释文档。当然，多行字符串也是支持的，而且遵循 Markdown 语法格式。例如：

```
"""                                # 注释开始
    bar(x[, y])                    # 原型简述

Compute the Bar index between `x` and `y`. If `y` is missing, compute
the Bar index between all pairs of columns of `x`.

# Examples                        # 一级标题
``julia-repl                     # 代码区开始，并紧随Markdown式的语言标识文字
julia> bar([1, 2], [1, 2])
1
``                                # 代码区结束
"""                                # 注释结束

function bar(x, y) ...
```

其中，先简单地给出函数的原型，再详细描述了函数的功能，并用 Markdown 给出了使用示例。

① Markdown 是一种语法简洁的标记语言，基于纯文本便可创建良好排版的文档；支持章节标题、着重强调、项目列表、表格、脚注引用、图片嵌入、HTML 嵌入、LaTeX 等。也能方便地转为 PDF 等其他格式。

Julia 这种文档字符串方式的注释内容因为与被注释对象紧邻，所以两者之间的注释关联会被编译器自动发现，成为内部文档系统的一部分。一旦代码被加载，便可在 REPL 或 IJulia 的帮助模式下查看创建的注释文档。

上例通过 `include()` 将脚本加载后，在 REPL 中查看的效果如图 16-1 所示。

```
help?> bar
search: bar baremodule SubArray GlobalRef clipboard BitArray backtrace

bar(x[, y])          # 注释开始
                    # 原型简述

Compute the Bar index between x and y. If y is missing, compute the
Bar index between all pairs of columns of x.

Examples             # 一级标题
=====

julia> bar([1, 2], [1, 2])
1

                    # 代码区结束
```

图 16-1 Markdown 方式的函数注释示例

在编写多行文档字符串时，用于界定的三引号 `"""` 标识符可单独占一行，这样能使文本格式更加清晰，即：

```
"""
...

...
"""
f(x, y) = ...

而不是：

"""...

..."""
f(x, y) = ...
```

另外，在写文档字符串时，也需留意每行的字符数量。过长的行会给阅读带来较大的困难。Julia 建议遵循代码同样的编写限制，一般以每行 92 个字符为宜。

在对 Julia 函数注释时，一般建议注释的内容包括以下几个方面：原型描述，简述或详述，参数描述列表，示例代码，公式或函数引用等。关于各段内容的描述方法，建议如下。

1. 原型描述

在文档字符串的头部给出函数的原型描述，简洁地说明函数的调用方法。可以缩进四个空格，这样在生成文档时，该原型描述能以 Markdown 方式被渲染成代码块。

当原型存在可选参数时，描述中尽可能给出完整的定义语法，例如，可直接写 `f(x, y=1)`；如果参数没有默认值，可用方括号表示，例如，`f(x[, y])` 或 `f(x[, y[, z]])`。

这种表述看起来会有些困难，可以采用多行方式列出可用的调用原型，例如：

```
f(x)
f(x, y)
f(x, y, z)
```

该方式同样可以用于列举函数的多个实现方法。

如果函数存在键值参数，只需在原型描述中使用 `<keyword arguments>` 作为占位符即可，例如 `f(x; <keyword arguments>)`，然后在 `# Arguments` 一节中列出每个键值参数的详细说明。

2. 简述与详述

在原型描述之后可给出简述，使用一两行语句简单地描述函数的功能或者用途，以及做什么并返回什么。如果需要，在简述之后隔开一个空行，给出详细的描述，而且可以分成多段进行阐述。但是参数描述不用放在详述中，因为有专门的方式。

描述时尽量简洁，不需要过多的衔接转换用词，只需直接描述要点即可。

3. 参数描述列表

只在真正需要的情况下提供参数的描述列表，对于简单的函数或者参数名称已经明了地表达了参数的意义时，再在注释中描述只是信息的重复而已，没有必要。但对于复杂的函数，提供参数列表还是必要的，尤其是存在键值参数的时候。

可在文档字符串中使用 `# Arguments` 标注开辟专门的一节来描述参数。之后在 Markdown 项目列表中每行描述一个参数：使用反引号 `` 给出参数原型（包括名称、类型、默认值等），以及参数的作用及意义，还可以包括参数的取值约束等等。例如：

```
"""
# Arguments
- `n::Integer`: the number of elements to compute.
- `dim::Integer=1`: the dimensions along which to perform the computation.
"""
```

其中，井号 `#` 前缀实际创建了 Markdown 一级标题，反引号对是行内代码标识符。

4. 示例代码

可在 `# Examples` 一节中以 `doctest` 的方式给出示例代码。所谓 `doctest`，是一个独立的代码块，以 ``jldoctest 标注，并包含 `julia>` 提示符，给出所有的输入及预期的输出，内容类似于 Julia REPL 的运行过程，就像在 REPL 打印出执行过程一样：

```
"""
# Examples
``jldoctest
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
# 代码块开始
# 模拟输入
# 模拟输出
```



```

3 4
...      # 代码块结束
"""

```

需要注意的是, 其中的 `doctest` 代码块部分实际完整地模拟了 REPL 的运行过程, 所以对格式非常敏感, 任何未能对齐的空格都会影响 `doctest` 的检验结果。另外, 因为 `doctest` 中 `julia>` 提示符之后的命令会在检验时运行, 所以应尽量避免随机数这种不稳定的函数调用; 与操作系统机器字节数相关的操作也会影响 `doctest` 的正确验证, 应尽量避免。

这种因故示例代码只能是不稳定的 (每次运行结果都会不同), 则可改用 ````julia` 进行标注, 这样能够在文档中获得正确的代码块。

5. 公式或函数引用

Julia 的标识符与代码片段应尽量使用反引号, 这样能够强调显示生成的说明文档。若在其中使用 LaTeX[Ⓔ] 编写公式, 应使用双反引号, 并尽量采用 Unicode 字符, 而不是使用 TeX[Ⓔ] 自己的转义字符, 例如应采用 ```α = 1``` 代替 ```\\alpha = 1``` 这种写法。

关于 Julia 文档系统、LaTeX 等方面更为详细的内容, 限于篇幅, 本书不再赘述, 读者可参考相关资料。

16.2 高性能编程建议

每个语言有其设计理念, 也有其独有的特点或者说是优劣势。在使用某种语言进行开发时, 除了我们在意的开发效率之外, 实际代码的运行效率也影响着我们对开发语言的选择。在时效性方面, 除了语言本身的特点影响之外, 对语言的使用也是极为重要的影响因素。同样的功能, 不同的人实现的方式不同, 在时效性方面也会存在极大的差异。熟悉语言, 掌握语言的优劣势, 才能更好地发挥语言的长处, 创建出稳定、高效的可执行程序。

Julia 中提供了很多可以对代码运行效率进行检测的方法, 例如, 10.3.3 中介绍的宏, 其中的 `@time` 能够很简洁地获得代码运行效果, 例如:

```

julia> function f(n)
    s = 0
    for i = 1:n
        s += i/2
    end
    s
end

```

-
- Ⓔ LaTeX 是由 Leslie Lamport 在 20 世纪 80 年代开发的排版系统, 在 TeX 的基础上扩展了很多更强大的功能, 比 TeX 更为易用。其官方网站为 <https://www.latex-project.org>
 - Ⓔ TeX 是由 Donald E. Knuth (高德纳) 发明的排版系统, 有 900 多条指令, 功能强大而且非常灵活, 可以输出专业级的高质量文档。它在学术界十分流行, 特别是数学、物理学和计算机领域, 是公认的数学公式排得最好的系统, 被许多世界一流的出版社如 Kluwer、Addison-Wesley、牛津大学出版社等用来出版书籍和期刊。AMS-TeX 与 LaTeX 系统等都是基于它的著名扩展。

```
f (generic function with 1 method)
```

```
julia> @time f(1);
0.012686 seconds (2.09 k allocations: 103.421 KiB)
```

```
julia> @time f(10^6)
0.021061 seconds (3.00 M allocations: 45.777 MiB, 11.69% gc time)
```

上例中，在第一次调用，即 `@time f(1)` 语句执行时，函数 `f()` 会进行编译，同时首次使用的 `@time` 也需要编译，所以首次的运行报告仅能作为参考。为了获得更为准确的报告，最好能重复地多次运行，即 `@time f(10^6)` 语句。

如果报告中出现了不可预期的内存分配，基本上都是代码存在某些问题的一个征兆。通常来说，是 Julia 中类型稳定性方面出了问题，导致生成的代码不是最佳的。为了能够更为准确地定位问题，可以使用 `BenchmarkTools.jl` 等包对代码进行更为严格的 Benchmark，以能够在代码优化等方面获得助益。

除此之外，包 `Profiling.jl` 也可以对运行的代码进行性能评估，并能识别出代码是否存在性能瓶颈。对于复杂的项目，包 `ProfileView.jl` 可以在性能可视化方面提供帮助。也可以在启动 Julia 时，加上 `--track-allocation=user` 选项，并检查 `*.mem` 为扩展名的文件，查看关于内存管理方面的信息，对内存分配进行深入的分析。而 `@code` 宏则能在代码编写方面生成一些数据，帮助找到类型声明不确切的低效率表达方式。还有一个 `Lint` 包，可以在代码编写过程中，对一些语法错误进行提醒。

通常，在代码实现中，除了必要的数据处理、业务逻辑之外，还有涉及内存分配等计算资源管理等操作，这些非核心的工程性过程往往会让开发变得极为复杂。在编程原则中，一般建议将不同性质的过程进行分离，比如，将处理逻辑与内存管理分开实现，从而能够在性能提升方面有所帮助，还可以使代码结构更为清晰，也更易于维护。

下文在如何优化代码设计方面提供了很多的建议，对于优秀的开发者而言，这些建议都是非常重要的，将其纳入自己在 Julia 语言方面的开发原则之中，必然能够获益良多，开发出优秀的基于 Julia 语言的系统。

16.2.1 类型

1. 少用全局性变量

在全局域内声明的变量，因为可能在任意地方改变取值甚至类型，所以是一种全局性变量。但全局域并不像局部域那样有着明确的边界，所以这种全局性变量发生变化的范围难以评估，导致编译器无法生成高效的代码，为性能优化带来了很大的麻烦。

笔者给出如下三个建议：

- ❑ 尽可能将这种全局性变量转为局部变量使用，通过函数封装的方式进行数据传递。
- ❑ 如果确定某个全局性变量作为全局变量使用，明确地以 `global` 关键字标识。例如。

```
global x
y = f(x::Int + 1)
```

□ 如果全局变量是常量，明确地以 `const` 关键字标识。例如 `const DEFAULT = 0`。

虽然 Julia 的 `const` 适用于局部变量和全局变量，但一般用于全局变量。使用 `const` 标识那些不会变的全局变量，编译器可以有针对性地对其优化，让程序在性能上受益。局部变量一般在代码块中出现，因为仅在局部，所以编译器能够自动判断它们的变更情况，所以无须为了性能将局部变量标记上 `const` 关键字。

2. 容器元素少用抽象类型

先看下面一段代码：

```
a = Real[]          # 数组a的类型为Array{Real,1}
if (f = rand()) < .8
    push!(a, f)
end
```

该例创建了数组，元素类型声明为抽象类型 `Real`；生成的随机数小于 0.8 时便将其放入其中。

我们知道，`Real` 的子类型有很多，包括各种浮点型、整型、有理数型等，有着互不相同的内存结构，甚至字节数都不同。在底层实施中，数据在放入数组 `a` 这种被要求容纳多种类型的容器时，只能单独地进行内存分配；而容器内元素真正记录的仅是该数据的引用。

实际上，上例中需要放入容器的都是同一种类型的浮点值，所以完全可以使用具体的元类型 `Float64` 创建该数组，即：

```
a = Float64[]      # 数组a的类型为Array{Float64,1}
```

此时，内部实施的过程将与上例完全不同：编译器会创建一块连续的内存区，且直接在其中存取元素数据，不需要额外的内存管理及引用操作。

显然，后一种方式比前一种方式有着更高的性能与效率。



建议 容器中尽可能采用具体类型，少用抽象类型；不在必需的时候为了灵活适用而牺牲效率。

3. 成员变量少用抽象类型

先定义一个非常简单的复合类型：

```
struct Simple
    v
end
```

其中，唯一成员 `v` 默认为 `Any` 类型。

下面我们以三个完全不同的数据创建 `Simple` 对象：

```
julia> a = Simple("Hello");
julia> b = Simple((1,2,3));
julia> c = Simple(1.5);
```

```
julia> typeof(a)
Simple
```



```
julia> typeof(b)
```

```
Simple
```

```
julia> typeof(c)
```

```
Simple
```

可见，虽然 a、b、c 提供的数据截然不同，但创建的对象类型却都是 Simple，而且 `typeof()` 的结果中没有任何附加的信息说明这三者间存在何种差异。

显然，三者中 `v` 的内存结构是完全不同的，编译器只能分析提供的值以进行代码生成。此外，在执行对 `v` 的计算中，CPU 甚至会采用完全不同的指令集。所以，这种类型信息的缺失，无论是在编译期还是运行期，都只能带来低效的性能。



建议 在创建或使用复合类型时，提供越多、越明确的类型信息，对编译器的代码优化和运行效率越有帮助。

不过，在提供成员的类型信息时，下面的 TA 方式：

```
mutable struct TA{T<:AbstractFloat}
    v::T
end
```

要优于下面的 TB 方式：

```
mutable struct TB
    v::AbstractFloat
end
```

分别用这两种方式创建对象，如下：

```
julia> a1 = TA(1.5)
```

```
TA{Float64}(1.5) # 创建具体类型为Float64的复合类型
```

```
julia> b = TB(2.5)
```

```
TB(2.5)
```

可见，返回的对象类型略有差异，前者信息更多些。再查看两者内部成员 `v` 的类型：

```
julia> typeof(a1.v)
```

```
Float64
```

```
julia> typeof(b.v)
```

```
Float64
```

可见，都是提供数值的默认浮点型 Float64。我们尝试以别的浮点型改变 `v` 的值，如下：

```
julia> a1.v = 1.1f0;
```

```
julia> b.v = 2.1f0;
```

```
julia> typeof(a1.v)
```

```
Float64
```

```
julia> typeof(b.v)
```

```
Float32
```

可见，类型 TA 的对象 `a1` 中的成员 `v` 的取值类型仍保持不变，而 TB 类型 `b` 的成员 `v` 除了

值外，类型也被同时改变了。

这是因为在创建对象时，在 TA 方式中编译器会根据 v 值的具体类型同时确定 TA 的 struct 类型，而一旦确定便不会再变更。但 TB 方式因为 Julia 弱类型机制的存在，导致成员类型会发生变化。

显然 TA 无论是在类型信息提供方面还是在内部类型稳定性方面，都要比 TB 方式有优势。在编译器生成内部代码时，前者能够获得更高的效率。



建议 在定义复合类型时，尽可能通过 struct 的类型参数化限定内部成员的类型；并在限定类型时，尽量采用具体类型而不是抽象类型。

需要特别注意的是，为第一种方式的 struct 类型创建对象时，如果强制成员 v 采用抽象类型 AbstractFloat，构造的 struct 本身将不再是具体类型，即：

```
julia> a2 = TA{AbstractFloat}(1.6)
TA{AbstractFloat}(1.6)      # 创建抽象类型为AbstractFloat的复合类型

julia> typeof(a2.v)
Float64

julia> a2.v = 1.7f0;
julia> typeof(a2.v)
Float32
```

可见，显式要求 struct 类参为 AbstractFloat 抽象类型时，创建的 TA 对象类型不再因为 1.6 是具体类型 Float64 而成为对应具体类型的 struct。虽然内部成员 v 在初始时是 Float64，但随后便被改变，因为 Float32 仍是 AbstractFloat 的子类型，struct 无法再限定内部成员一定要为某个具体类型。这是 Julia 内部实现中不可忽视的细节。



建议 参数化复合类型创建对象时，在不必要的情况下，不要显式地限制类型参数的选择，尤其不要强制使用抽象的类型参数。

4. 成员变量少用抽象容器

下面我们使用两种类似的方式定义复合类型，其中的唯一成员为数组类型，如下：

```
mutable struct CA{S <: AbstractArray{T,1} where T}
    v::S
end

mutable struct CB{T}
    v:: AbstractArray{T,1}
end
```

分别进行实例化，例如：

```
julia> a1 = CA{1:3}
CA{UnitRange{Int64}}(1:3)

julia> a2 = CA{[1:3]}
CA{Array{Int64,1}}([1, 2, 3])
```

```
julia> b1 = CB{5:8}
CB{Int64}(5:8)

julia> b2 = CB{[5:8;]}
CB{Int64}([5, 6, 7, 8])
```

显然，前一方式创建的对象有着更为丰富的 struct 信息。依照之前的经验总结，类型信息更为丰富的第一种方式能够获得更好的性能与效率。

道理与前文所述类似，虽然成员 v 的类型可以是任意的向量类型，但在对象创建时，CA 会以实际提供 v 的具体类型对 struct 进行实例化，所以会生成一个类型具体的复合类型；而 CB 的 struct 只能体现 v 元素的类型，无法告知编译器 v 到底是怎样的向量结果。



建议 在定义复合类型时，尽可能采用类型具体的容器作为成员；而且尽可能让内部类型信息暴露在 struct 层面，这样在 struct 生成代码或进行操作时会具有更高的效率。

16.2.2 函数

1. 明确标识键值参数的类型

函数在调用时参变量的类型会在函数体执行前落实为具体类型，所以定义函数时是否明确地限定参数类型不会直接影响内部语句的性能表现。但明确的参数类型标识，能够减少调用动作的消耗，尤其是存在键值参数的时候。所以，最好能够限定键值参数的类型，例如：

```
function with_keyword(x; name::Int = 1)
    # 实现体
end
```

对于包含键值参数的函数，如果调用方仅使用了有序实参而未提供键值实参，则不会有什么调用消耗；但如果调用方提供了键值实参，便会出现消耗，因为键值参数是无序的、动态的，存在解析成本。



建议 定义函数时，尽可能不使用键值参数，尤其是在性能敏感的代码中；一定要使用时，提供明确的类型标识，而且最好是具体类型。

2. 返回类型稳定一致

假设有一个简单的函数，但需要使用抽象的类型参数，如下：

```
pos(x::Real) = x < 0 ? 0 : x
```

其中， x 可能是整型、浮点型或有理数型。尝试作如下调用：

```
julia> typeof(pos(Int8(1)))
Int8
```

```
julia> typeof(pos(1//2))
Rational{Int64}
```

```
julia> typeof(pos(1.1))
```

```
Float64
```

```
julia> typeof(pos(Int8(-1)))
Int64
```

```
julia> typeof(pos(-1//2))
Int64
```

可见, 当实参值大于零时, `pos()` 函数返回的类型会随着实参类型的变化而变化, 只在小于零时, 返回类型才会一直为函数中字面值 0 的默认 `Int64` 类型。而且, 即便 `x` 的类型限定为某个具体的非 `Int64` 类型, 不同的 `x` 值也会导致返回类型的不稳定。

虽然这种不断变化的类型不会对 `pos()` 本身的效率带来太多的影响, 但如果在代码的调用处, 将 `pos()` 的返回值作为输入, 便会出现类型不明确的问题。例如:

```
julia> fa(x) = zeros(typeof(pos(x)), 3);
```

显然, `zeros()` 创建的三阶向量的元素类型只有在 `x` 确定时才能够确定下来。

所以, 在开发过程中, 尽可能让函数的返回值不发生变化, 这样不仅有助于代码质量的控制, 也有助于性能的优化。

上例可以改进成以下的方式:

```
pos(x::Real) = x < 0 ? zero(x) : x
```

这样即便要返回 0 值, 类型也会与 `x` 的类型一致。



建议 定义函数时, 尤其是存在多个返回分支的时候, 应尽可能确保返回值类型的稳定, 必要时可以借助 `zero()` 或 `one()` 函数。

3. 类型不随意变更

类型的稳定性问题也会在某个变量被反复使用的情况下出现。例如:

```
function foo()
    x = 1
    for i = 1:10
        x = x/bar()
    end
    return x
end
```

其中, 变量 `x` 首先是整型 (`Int64`), 但在循环第一次执行时便因为除法操作变成了浮点型。这种情况会导致编译器优化内部循环结构时出现很大困难。

上例可以通过以下方式进行优化:

- ❑ 初始化 `x` 为浮点数值, 即 `x=1.0`;
- ❑ 声明 `x` 的类型为浮点型, 即 `x::Float64 = 1`;
- ❑ 使用显式的转换, 即 `x = oneunit(Float64)`;
- ❑ 剥离出第一层循环, 在单独的语句中让 `x` 类型发生转换, 然后再执行后续的循环, 即 `for i = 2:10`。

在开发中要尽可能避免这种类型变更的情况出现。



建议 在计算过程中，尽量让变量与类型的绑定关系稳定，不随意变更其类型。

4. 定义小而精的函数

将常用语句进行封装为函数，不但有利于功能复用与灵活控制，而且可使代码结构、运行逻辑更为清晰。另外，在 Julia 中，函数是一个较为封闭的代码块，有着目前的作用域边界，也有利于编译器生成精简高效的代码。



建议 函数应尽可能短小精悍，内部功能单一、独立，不要混杂过多的处理逻辑。

例如下面的函数：

```
function norm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return maximum(svd(A).S)
    else
        error("norm: invalid argument")
    end
end
```

完全可以拆分成两个更为简洁、清晰的同名方法：

```
norm(x::Vector) = sqrt(real(dot(x,x)))
norm(A::Matrix) = maximum(svd(A).S)
```

对于这种短小的函数，编译器会自动内联 (inline)，以减少编译及运行消耗。

5. 隔离核心计算过程

先看一个例子，如下：

```
function strange_twos_1(n)
    a = Vector{rand{Bool} ? Int64 : Float64}(undef, n)
    for i = 1:n
        a[i] = 2
    end
    return a
end
```

该函数会先根据随机布尔值生成元素类型为 Int64 或 Float64 的 n 维向量，然后再通过 for 循环对数组的内容进行赋值操作。

遵循应尽量采用短小精悍函数的经验，将上例修改为以下两个函数：

```
function fill_twos!(a)
    for i=1:length(a)
        a[i] = 2
    end
end
```



```
function strange_twos_2(n)
    a = Array{rand{Bool} ? Int64 : Float64}(undef, n)
    fill_twos!(a)
    return a
end
```

将参数 n 设为 100000000，分别调用 `strange_twos_1()` 与 `strange_twos_2()` 函数，并统计运行效果，如下：

```
julia> @time strange_twos_1(100000000);
0.358861 seconds (10.00 M allocations: 228.874 MiB, 36.11% gc time)

julia> @time strange_twos_1(100000000);
0.393932 seconds (10.00 M allocations: 228.874 MiB, 32.23% gc time)

julia> @time strange_twos_2(100000000);
0.031758 seconds (14 allocations: 76.295 MiB, 2.09% gc time)

julia> @time strange_twos_2(100000000);
0.029737 seconds (14 allocations: 76.295 MiB, 2.22% gc time)
```

可见，两个版本的表现差异极大。无论是运行时间、内存分配还是垃圾回收方面，第二版的函数 `strange_twos_2()` 都有着极为明显的改善。

实现中主要有两个基本步骤：数组的创建及其元素的更新。在第一版中，编译器会同时优化这两个步骤，但因为一时无法确定 a 的类型，所以难以对循环体进行深入的优化；但在第二版中，在处理 `fill_twos!()` 时数组的类型已经是确定的，所以单独处理该函数中的循环体能够生成更高效的代码。



建议 将核心计算过程剥离到独立的函数中，以构成函数围栏（function barriers），让编译器能够逐层、逐步地优化，使得程序的整体性能获得大幅度提高。

事实上，不同功能的过程分离在不同的函数或接口中，也是典型的编程规范。不但能够让结构更为清晰，也有利于模块化处理。尤其是在团队合作的大型工程中，这一点尤为重要。

6. 避免多态分发的误用

类型参数化与多态分发机制让 Julia 程序变得极为灵活，开发也极为高效快捷。但如果过度使用会带来糟糕的后果。

假设定义了如下的复合类型：

```
struct Car{Maker,Model}
    year::Int
    # 更多字段
end
```

然后在其基础上开发各种功能，包括函数、数组等。由于类型参数 `Maker` 与 `Model` 的存在，这其中必然会涉及多态分发问题。

如果需要遍历一个以 `Car` 为元素的数组，并调用定义的函数对元素做处理，这时编译

器只能在运行期确定具体的类型，在方法列表中查询合适匹配的实现方法，并决定是否需要进行 JIT 编译，然后才实施调用执行操作。实际上，这已相当于是完整的类型标定与 JIT 编译过程，但却是在运行期完成的，所以程序会变得极为缓慢。

但是受这种“类型组合爆炸”影响更多的或许还是编译期。在底层，针对每个不同的 `Car{Make, Model}`，Julia 会编译出特定的函数（specialized functions）。但如果这种类型成百上千，那么以其为参数的函数便会有成百上千的变体，导致编译结果缓存，内部方法列表都会大大增加，带来资源的极大消耗。

如果存在以下两种情况，才建议考虑这种方式：

- ❑ Car 对象的处理是 CPU 敏感的；在 Make 和 Model 明确时，有较好的性能表现。
- ❑ 大量的同性质 Car 待处理，使用数组更方便，例如 `Array{Car{:Honda,:Accord},N}`。

在后一种情况中，容器中所有元素类型统一，所以编译器可提前知道每个元素的类型，能够在函数编译时找到匹配的方法，并生成高效的运行代码。



建议 恰当地使用类型参数化与多态分发机制，可以发挥 Julia 语言强大的优势；但不能过度地依赖，更不能滥用。

16.2.3 数组

1. 预先分配结果内存

先看实现相同功能的两个不同方式，如下：

```
function xinc(x)
    return [x, x+1, x+2]
end
```

基于输入对象创建3元素数组并返回

```
function loopinc()
    y = 0
    for i = 1:10^7
        ret = xinc(i)
        y += ret[2]
    end
    y
end
```

取xinc返回数组中第2个元素累加

返回累加结果

以及：

```
function xinc!(ret::AbstractVector{T}, x::T) where T
    ret[1] = x
    ret[2] = x+1
    ret[3] = x+2
    nothing
end
```

修改已有的数组对象

避免函数返回值

```
function loopinc_prealloc()
    ret = Array{Int}(undef, 3)
    y = 0
    for i = 1:10^7
        xinc!(ret, i)
    end
    y
end
```

循环之前预先创建3元素的数组

```

        y += ret[2]
    end
    y
end
# 返回累加结果

```

分别调用并比较执行效果：

```

julia> @time loopinc()
0.529894 seconds (40.00 M allocations: 1.490 GiB, 12.14% gc time)
50000015000000

julia> @time loopinc_prealloc()
0.030850 seconds (6 allocations: 288 bytes)
50000015000000

```

可见，结果虽然一致但耗时相差近 17 倍，内存分配与垃圾回收方面更是天壤之别。后者基本不占用内存，但前者却前前后后占用了近 1.5G 内存，而且还需要垃圾回收。

以上两种实现方式最大的差别在于，`xinc` 提供三元数组时，是否每次都要重新创建。在第一种方式中，`xinc` 返回数组结构时，在底层实际要分配内存并新建数组对象；但 `xinc` 是在循环内部被调用的，导致这种创建分配过程被高度频繁地执行，随之而来的便是内存的释放与垃圾回收操作。第二种方式只预先分配一次内存，仅创建一次数组对象，并通过引用传递的方式更新其中的值，所以效率更高。不仅如此，第二种方式更方便对输出结果的类型进行控制。



建议 尽可能少的内存操作也是提升性能的关键；在对大规模或复杂数据进行处理时，应尽可能避免频繁的内存分配、释放与垃圾回收，尤其是不要在循环内部涉及内存操作。

实际上，Julia 中点操作形式的矢量版函数能够融合多重循环，且能够进行就地修改运算。恰当地使用能够避免频繁的内存操作，高效地达到预期目的。

2. 多用点操作进行矢量化运算

对数组进行相同的逐元运算时，往往需要对其遍历循环。Julia 的点操作能够很方便地将任意的“变量”函数转为“矢量”版，在内部自动构造循环结构，实现逐元的函数调用；当表达式中存在多个点操作时，能够自动将多重逐元操作进行融合，无须多次循环；而且可就地修改原数组，无须分配临时数组对象。即：使用 `vector.+vector` 与 `vector.*scalar` 要优于使用 `vector+vector` 与 `vector*scalar` 这种方式（其中 `vector` 指某数组，`scalar` 指某标量值）。

例如，有以下两种功能一致的函数：

```

f(x) = 3x.^2 + 4x + 7x.^3
fdot(x) = @. 3x^2 + 4x + 7x^3    # 等效于 3 .* x.^2 .+ 4 .* x .+ 7 .* x.^3

```

但执行效果并不相同：

```

julia> x = rand(10^6);

```



```
julia> @time f(x);
0.010986 seconds (18 allocations: 53.406 MiB, 11.45% gc time)

julia> @time fdot(x);
0.003470 seconds (6 allocations: 7.630 MiB)

julia> @time f.(x);
0.003297 seconds (30 allocations: 7.631 MiB)
```

其中, 函数 `fdot()` 的用时只是 `f()` 的 1/3, 而且分配的内存只有 `f()` 的 1/7。



建议 应尽可能使用点操作对数组进行矢量化计算。

3. 按列访问数组

如前文所述, Julia 中的多维数组是按列存储的。所以在对数组线性访问时, 元素索引值的第一维 (行索引) 会变得更快速。例如:

```
julia> a = [1 2; 3 4; 5 6]
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

```
julia> a[:]          # 将数组转为向量
6-element Array{Int64,1}:
 1          # 行1列1
 3          # 行2列1
 5          # 行3列1
 2          # 行1列2
 4          # 行2列2
 6          # 行3列2
```

高维数组在转为向量后遍历时, 元素对应的原索引中行号要比列号变得更快, 即更不稳定。

所以通过多维索引的方式访问高维数组时, 列索引迭代处于遍历表达式的最内层时, 能够获得更快的访问效率, 因为无须在内存中进行不断的跳跃。

下面我们通过实例进行验证: 假设创建一个方阵, 其元素值由另一向量按行或按列复制所得, 以四种不同的方式实现:

```
function copy_cols(x::Vector{T}) where T
    n = size(x, 1)
    out = Array{T}(n, n)
    for i = 1:n
        out[:, i] = x
    end
    out
end
```

```
function copy_rows(x::Vector{T}) where T
    n = size(x, 1)
    out = Array{T}(n, n)
    for i = 1:n
        out[i, :] = x
    end
end
```



```

    end
    out
end

function copy_col_row(x::Vector{T}) where T
    n = size(x, 1)
    out = Array{T}(n, n)
    for col = 1:n, row = 1:n
        out[row, col] = x[row]
    end
    out
end

function copy_row_col(x::Vector{T}) where T
    n = size(x, 1)
    out = Array{T}(n, n)
    for row = 1:n, col = 1:n
        out[row, col] = x[col]
    end
    out
end

```

然后输入一个长度为 10000 的向量作为参数，分别调用评估，如下所示：

```

julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed f(x))

julia> map(fmt, Any[copy_cols, copy_rows, copy_col_row, copy_row_col]);
copy_cols:      0.331706323
copy_rows:      1.799009911
copy_col_row:   0.415630047
copy_row_col:   1.721531501

```

可见，`copy_cols()` 比 `copy_rows()` 要快得多，这是因为 `copy_cols()` 利用了列式存储的内存结构，能够一次性将整列复制过去。另外，函数 `copy_col_row()` 要比 `copy_row_col()` 快，这是因为两层循环中，列索引处于遍历表达式的最内层，有着更高的存取性能。



建议 在对高维数组进行存取操作时，最好遵循其内部的存储结构，以便获得更高的性能。

4. 多采用数组视图

在第 8 章中介绍过，范围切片可用于取得子数组，能够作为左操作数直接修改原数据中的内容，例如，`a[1:3, :] = 1.0` 会将对应元素全部更新为 1.0；但作为右操作数时，会创建新数组（内存分配），并将数组复制到新的数组中，例如 `b = a[1:3, :]`。

在实际开发中，如果对切片生成的新数组计算很少，这种操作是得不偿失的。



建议 尽量基于原数组的视图进行处理；不但能够就地更新，而且不会有额外的内存操作。

16.2.4 IO

1. IO 时避免字符串展开

Julia 的字符串支持将其中以 \$ 标识的变量进行展开，但这样会生成临时字符串。



建议 在不必要时，尽量不要在字符串中使用变量展开功能，尤其是在涉及文件写入等 IO 操作的场景中。

例如最好用 `println(file, a, " ", b)` 代替 `println(file, "$a $b")` 这种方式，因为前者会直接将变量 `a` 与 `b` 的值写入 `file` 文件。而且过多的字符串内变量也会导致代码的可读性变差，例如 `println(file, "$(f(a))$(f(b))")` 最好用 `println(file, f(a), f(b))` 替代。

2. 优化并发中的网络 IO

在跨机器的远程调用中，会涉及网络通信，而网络 IO 操作是极为耗时的过程。不同的实现方式会在这方面存在较大的差异。例如：

```
responses = Vector{Any}(nworkers())
@sync begin
    for (idx, pid) in enumerate(workers())
        @async responses[idx] = remotecall_fetch(pid, foo, args...)
    end
end
```

要优于：

```
refs = Vector{Any}(nworkers())
for (idx, pid) in enumerate(workers())
    refs[idx] = @spawnat pid foo(args...)
end
responses = [fetch(r) for r in refs]
```

因为第一种实现中每个 Worker 只需进行一次网络交互，但后一种则需两次：第一次是 `@spawnat`，而第二次则是 `fetch` 操作。而且后一种实现中，`responses` 采用了串行的方式获取远程引用，会导致整体性能的严重降低。



建议 在远程调用时，尽可能减少网络通信，最好能够将耗时操作以异步的方式并行化。

16.2.5 其他

1. 恰当地使用注解宏

在开发中，可以通过注解（Annotations）宏对程序的运行属性进行设定，避免不必要的底层操作，从而实现性能上的优化。

例如，可使用 `@inbounds` 宏避免数组的边界检查：

```
function inner(x, y)
    s = zero(eltype(x))
```

```

for i=1:length(x)
    @inbounds s += x[i]*y[i]
end
s
end

```

当然实施此操作时，需要确保语句运行时真的不会出现越界错误，否则程序的悄然崩溃真的会令人“崩溃”。

再比如，可使用 `@fastmath` 宏对浮点数操作进行优化，但会让结果与 IEEE 的数值标准有所出入；还有一个 `@simd` 宏，能自动将 `for` 循环矢量化，但此功能仍处于实验阶段，不建议使用。



建议 在性能要求极为严格时，恰当使用注解宏有助于性能的提升，但需要慎重。

2. 不用已弃用功能

在版本的快速更迭中，总会有些功能或函数被弃用（deprecated）。如果在开发中调用了弃用的函数，编译器为了打印警告信息，内部会执行必要的查找操作。这种与业务代码无关的查找与打印过程会严重地拖慢程序的运行。



建议 所有被告知已经弃用的调用，都应依照警告建议尽可能地修正。

3. 注意更多的性能细节

在开发中，一些细节也会对性能产生影响，最好能够避免，包括：

- ❑ 避免使用不必要的数组，例如应使用 `x+y+z` 代替 `sum([x,y,z])` 这种较重的表述。
- ❑ 计算复数模时，应使用 `abs2(z)` 而不是 `abs(z)^2`。
- ❑ 获得整型除数，应使用 `div(x,y)` 代替 `trunc(x/y)`，因为后者返回的是浮点类型。
- ❑ 使用 `fld(x,y)` 代替 `floor(x/y)`；使用 `cld(x,y)` 代替 `ceil(x/y)`。



建议 多熟悉 Julia 提供的各种库函数，并尽量使用其内置的方法，因为它们往往已经过优化。

16.3 与其他语言的异同

在编写 Julia 代码的过程中，会发现与其他语言有不少相似之处，例如 Matlab、R 或 Python，但它们之间也有着明显的不同。通过与其他语言的对比，我们可以更加深入地理解 Julia，也能够熟悉其他语言的基础上，更快地学习 Julia 语言。

16.3.1 与 Python 相比

Python 语言与 Julia 同为动态语言，都有着极为高效简洁的开发风格，主要差异有：

- ❑ Julia 代码编写格式无要求。众所周知, Python 在代码编写时, 是需要“游标卡尺”的, 格式就是语法规则的一部分, 而 Julia 语言没有这方面的限制, 以关键字对 (以 `end` 结束) 形成构造完整的结构。
- ❑ Julia 中没有 Python 中的 `pass` 关键字。
- ❑ Julia 中对数组、字符串等的索引是从 1 开始的, 而不是像 Python 等语言那样从 0 开始。
- ❑ 数组 slice 索引时, Julia 取得最后的元素。在对数组进行 slice 索引时, Julia 按 slice 结构包括最后一个元素, 但 Python 最后一个元素并不在 slice 中。例如, Julia 中的 `a[2:3]` 取得第二个及第三个元素, 相当于 Python 中的 `a[1:3]`, 这个差别需要特别注意。
- ❑ Julia 不支持负值索引。在对数组索引时, Julia 不支持 Python 中以负值的方式按倒序取得数组中的元素。但 Julia 可以通过 `end` 关键字取得数组中的最后一个元素, 并可通过 `end-n` 获得倒数 `n` 个元素。
- ❑ Julia 数组采用列序存储, 而 Python 中的 NumPy 默认采用的是行序存储方式。
- ❑ 运算符 `%` 在 Julia 中是取余操作, 在 Python 中则是取模操作。
- ❑ Julia 更新运算符不会就地修改原变量。需要注意的是, Julia 中的更新运算符与 Python 中的有着很大不同, 包括 `+=`, `-=` 等, 不会就地修改原变量, 例如:

```
julia> A = [1 1 1 1]
1×4 Array{Int64,2}:
 1  1  1  1
```

```
julia> B = A;
```

```
julia> B
1×4 Array{Int64,2}:
 1  1  1  1
```

```
julia> B += 3
1×4 Array{Int64,2}:
 4  4  4  4
```

```
julia> B
1×4 Array{Int64,2}:
 4  4  4  4
```

```
julia> A
1×4 Array{Int64,2}:
 1  1  1  1
```

其中, A 没有被改变, 只是 B 被改变了。运算符 `+=` 只是相当于 `B=B+3` 操作, 在此过程中, `B=A` 建立的引用关系被解除, 变量名 B 被重新绑定到了 `B+3` 生成的新数组上。

- ❑ Julia 参数默认表达式总会重新求值。在函数采用默认参数值的情况下, 函数每次运行时, Julia 都会重新执行参数值中的表达式, 不像 Python 那样只运行一次, 这会带来很大的不同。例如在 Julia 中:


```
julia> f(x=rand()) = x
f (generic function with 2 methods)
```

```
julia> f()
0.8382948989948376
```

```
julia> f()
0.8835315579456324
```

```
julia> f()
0.32548534620112024
```

函数 `f()` 的每一次调用, 参数 `rand()` 都会被执行, 导致结果出现不同。而在 Python 中, 则有着完全不同的表现:

```
>>> import random
>>> def f(x = random.randint(0,100)):
...     print x
...
>>> f()
24
>>> f()
24
>>> f()
24
```

可见, 参数中虽然使用了 `random` 生成随机数, 但每次 `f()` 被调用, 返回的都是同一个值, 这是因为 Python 在函数被第一次调用时, 参数表中的表达式被计算并固化下来。

所以, 如果读者原来熟悉 Python, 转而学习 Julia 语言, 对这些不同点要非常小心, 否则很容易出现难以察觉的错误。

16.3.2 与 Matlab 相比

Julia 与 Matlab 有不少相似之处, 但在功能、句法等方面又有着很多的不同, 包括:

- ❑ Julia 中数组下标采用方括号, 而 Matlab 是圆括号。
- ❑ Julia 函数传参采用共享机制, 复杂结构是引用方式; 而 Matlab 是简单的传值。
- ❑ 数组赋值在 Julia 中只是将原数据绑定到了新的名字, 可以认为是新的引用, 所以不会像 Matlab 那样要新建数组并赋值数据。
- ❑ Julia 不会在对数组元素的赋值时自动增长数组。在 Matlab 中, `a(4) = 3.2` 会自动创建数组 `a=[0 0 0 3.2]`; 而 `a(5)=7` 时, 因 5 超出 `a` 的长度导致 `a` 被自动增长, 变成 `[0 0 0 3.2 7]`。但在 Julia 中, `a[5]=7` 对元素赋值时, 如果 5 超出 `a` 实际长度, 会抛出异常。若要对数组增加元素, Julia 则利用函数 `push!()` 或 `append!()`, 这种方法比 Matlab 中的 `a(end+1)=val` 更为高效。
- ❑ 复数虚部在 Julia 中的标识符为 `im`, 而 Matlab 采用 `i` 或 `j` 标识符。
- ❑ Matlab 默认字面值为浮点数, 但 Julia 会自动判断其最合适的类型。
- ❑ Julia 有真实的向量结构, 而不是通过矩阵模拟。
- ❑ 范围表达式在 Julia 中是独立类型。在 Julia 中, 表达式 `a:b` 或 `a:b:c` 会构造类型

为 Range 的独立对象，而不会像 Matlab 那样自动展开为向量。Julia 可使用 `collect()` 将 Range 对象变成数组，但在实际使用中一般不需要这种转换，因为在大部分情况下其使用与数组是一致的。而且 Julia 因为懒计算的特点，有着更高的性能。这种对数组有一定代替性的 Range 对象，在一些函数或迭代操作中极为常用。

- ❑ 两者返回结果的方式不同。Julia 函数会将最后一个表达式的结果返回，除非有显式的 `return` 调用。若结果是多个，Julia 会自动将其组装为 Tuple 类型；Matlab 则采用了 `nargout` 变量，描述返回值。
- ❑ Julia 在运行期间，没有严格的工作目录的概念。
- ❑ Julia 中排序函数 `sort()` 默认按列进行。Julia 对数组调用 `sort(A)` 等价于 `sort(A,1)`，如果要对 $1 \times N$ 数组的所有元素进行排序，使用 `sort(A)` 不会获得预期效果，要通过 `sort(A,2)` 才可以。
- ❑ 两者在数组比较运算时有所不同。在 Julia 中，如果 A 与 B 是数组，对它们的逻辑比较操作（如 `A==B`）则并不会返回布尔型的数组，给出每个元素的比较结果，而是要用点操作的方式，即 `A.==B`，才会得到这样的比较结果。
- ❑ 在 Julia 中，位运算符 `&`（与）、`|`（或）和 `⊕`（异或）分别等价于 Matlab 中的 `and` 操作符、`or` 操作符及 `xor` 操作符，但在操作优先级方面略有不同。
- ❑ Julia 会自动检测跨行的表达式或语句结构，并不采用省略符来连接跨行的表达式。
- ❑ 内置变量 `ans` 在 Matlab 的脚本中会有效，但在 Julia 的脚本中会被忽略。
- ❑ Julia 中声明的类型无法在运行期间像 Matlab 的 `class` 那样能够动态增加成员。如果涉及，Julia 中最好采用 Dict 类型代替。
- ❑ Matlab 中只有一个全局作用域，但在 Julia 中每个模块都有自己独立的全局作用域及命名空间。
- ❑ 两者在过滤数组元素的方式上有所不同。在 Matlab 中，若要移除数组中不需要的元素，需通过逻辑索引就地修改原数组，例如 `x(x>3)` 或 `x(x>3)=[]` 选择或去除值 >3 的元素。在 Julia 中，提供了专门的高阶函数 `filter()` 及 `filter!()` 用于达到此目的，例如可使用 `filter(z->z>3, x)` 或 `filter!(z->z>3,x)` 实现上述功能。而且此两个函数能够减少临时内存的使用，有着较高的性能。

16.3.3 与 R 相比

相比 R 语言，Julia 在性能方面有着明显的优势。两者在语法及使用上的差异主要包括：

- ❑ 单引号在 Julia 中用于表达某个字符值，而非是字符串类型。
- ❑ R 中要获得子字符串，需先将其转换为字符向量，但 Julia 中可直接以索引的方式取得。
- ❑ 取模操作在 R 中是 `a %% b`，在 Julia 中则是 `mod(a,b)`，`%` 用于求取余数。
- ❑ 创建向量时，Julia 的 `[1,2,3]` 等效于 R 中的 `c(1,2,3)`。
- ❑ Julia 中逻辑索引需原数组维度与阶数一致。在 R 语言中，`c(1, 2, 3, 4)` `[c(TRUE, FALSE, TRUE, FALSE)]` 或 `c(1, 2, 3, 4)[c(TRUE, FALSE)]`

均可获得 `c(1, 3)` 这样的结果。但在 Julia 中, `[1, 2, 3, 4][[true, false]]` 会抛出 `BoundsError` 异常, 而 `[1, 2, 3, 4][[true, false, true, false]]` 才会获得想要的结果。

- Julia 二元运算符操作数均为数组时, 维度与阶数需一致。在对两个数组操作时, Julia 一般要求它们的维度与阶数一致, 例如 `[1, 2, 3, 4] + [1, 2]` 这样的表述会抛出错误。而 R 语言只要求两者重叠即可, 即 `c(1, 2, 3, 4) + c(1, 2)` 是有效的。
- Julia 的 `map()` 函数与 R 中的 `lapply()` 类似, 但参数表述有差异。
- Julia 的矢量化操作更为简单。R 中的 `mapply(choose, 11:13, 1:3)` 类似于 Julia 的 `broadcast(binomial, 11:13, 1:3)`。而 Julia 提供了更为简洁的点运算, 即 `binomial.(11:13, 1:3)` 便可达到相同的效果。
- 运算符 `->` 在 Julia 用于定义匿名函数, 但在 R 中用于赋值操作。
- Julia 中没有 R 中的 `<-` 与 `<<-` 操作符。
- 在对向量与矩阵进行连接时, Julia 使用的是 `hcat()`、`vcat()` 及 `hvcat()` 函数, 对应于 R 中的 `c()`、`rbind()` 及 `cbind()` 函数。
- Julia 中矩阵乘法使用 `A*B`, R 中需使用 `A %*% B`。
- 矩阵逐元乘积, 在 Julia 中使用点运算 `A.*B` 实现, 在 R 中的则是 `A*B`。Julia 的 `A'` 等效于 R 中的 `t(A)`, 用于求解矩阵的转置, 而 `!` 操作符则用于共轭转置。
- 在 Julia 中, 使用 `if` 结构、`for` 及 `while` 时, 条件表达式不需要圆括号。
- Julia 中数值 1 或 0 不能当布尔型使用。
- 获得数组阶数, Julia 使用 `size()` 函数, R 使用 `nrow()` 及 `ncol()` 函数。
- Julia 的类型限定更为严格。在 Julia 中, 标量、向量及矩阵都是不同的, 但 R 语言中标量 1 与向量 `c(1)` 是相同的。
- Julia 中的 `diag()` 及 `diagm()` 与 R 的同名函数有不同的功能。
- 在 Julia 中, 函数结果不能直接作为赋值的左操作数, 例如 `diag(A)=ones(n)` 是错误的。
- Julia 有完善的自定义类型机制, 也更为灵活便捷, 不像 R 要借助 S3 或 S4 对象。
- Julia 有更高效率的参数传递机制。函数调用时, Julia 采用共享传参机制, 复杂类型通过引用方式传递。与 R 中的传值方式相比, Julia 显然有着更为高效的性能。
- Julia 中范围表达式有独立的类型。表达式 `a:b` 在 R 中是向量的简洁表述, 但仍是向量。但 `a:b` 在 Julia 中是独立的 `Range` 类型, 是一种迭代器, 不会自动展开为数组, 在节省内存方面有着很大的优势。
- 两者的最大最小等函数存在差异。Julia 的 `max()` 和 `min()` 函数类似于 R 中的 `pmax()` 及 `pmin()` 函数, 而 R 中的 `max()` 及 `min()` 对应于 Julia 中的 `maximum()` 及 `minimum()` 函数。
- 两者中矢量化计算的意义有差异。R 语言的代码如果要提升性能, 往往要进行矢量化改造, 但 Julia 中的矢量化计算多是一种方便的形式, 不是性能的关键。

16.4 Julia 代码风格

本节在 Julia 编码方面给出一些建议供读者参考，但不是绝对的、必须遵循的。

❑ 遵循 Julia Base 模块的取名惯例。命名规则可视为一种惯例，并无绝对与强制，为的是增加识别和可读性。一旦选用或设定好命名规则，在程式编写时应保持一致格式。

- 变量：采用小写，可用下划线连接不同单词，但尽量不用下划线。
- 模块与类型：采用驼峰式^①，即单词首字母大写，例如 `SparseArrays`，`Unit-Range`。
- 函数与宏：全部使用小写字母，例如 `maximum()` 或 `convert()`；多词构成的完整措辞也写在一起，例如 `isequal()` 或 `haskey()`；必要时，可使用下划线分割多个词，例如有多个概念时，如 `remotecall_fetch()`。但如果函数名需要过多词，建议考虑一下，是否该函数实现的功能太多，分拆成多个是不是更好。

❑ 多写些函数。虽然对解决问题来说，一步步地写出处理过程相当直截了当，但最好还是尽早将程序拆分为函数。函数可让代码结构更为清晰，更易于复用和测试，而且因为输入输出明确所以也更易于控制。此外，鉴于 Julia 编译器的原理，函数比粗陋的语句罗列运行效率更高。

❑ 函数操作的数据应以参数传入，而不是直接使用全局性变量（除了像 `pi` 这种常量）。

❑ 避免使用奇怪的类型联合。像 `Union{Function, AbstractString}` 这种声明，通常意味着程序设计还不够清晰，需要重新考虑。

❑ 避免过度详细或复杂的容器类型。例如：

```
a = Array{Union{Int,AbstractString,Tuple,Array}}(n)
```

太过繁琐，此时直接使用 `Array{Any}(n)` 要好得多。

❑ 不使用不必要的静态参数。例如函数原型：

```
foo(x::T) where {T<:Real} = ...
```

应被写为：

```
foo(x::Real) = ...
```

尤其是 `T` 并没有在函数体内使用的时候。即使 `T` 被使用了，也应在合适的时候使用 `typeof(x)` 代替。与性能无关。注意该建议仅针对类型参数未被使用的时候。

❑ 不要重载基础容器的方法。以下面的函数为例：

```
show(io::IO, v::Vector{MyType}) = ...
```

该函数希望为某个新类型 `MyType` 元素的向量提供内容打印功能。虽然吸引人，但应尽量避免。原因在于，众所周知的类型 `Vector` 有固有的操作方法，过度地自定义其行为，会导致该类型变得难以使用。

① 驼峰式一词来自 Perl 语言中普遍使用的大小写混合格式，而 Larry Wall 等人所著的畅销书《Programming Perl》封面图片正是一匹骆驼。

- ❑ 小心对待类型一致性判断。一般应使用 `isa()` 或 `<:` 进行类型断言，而不是 `==` 操作符。一般而言，检查类型是否确切相等，仅在与已知的实际类型比较时才有意义，例如 `T==Float64`。
- ❑ 为修改了参数的函数附上 `!` 标记。例如函数：

```
function double(a::AbstractArray{<:Number})
    for i = 1:lastindex(a)
        a[i] *= 2
    end
    return a
end
```

修改了参数中的数组 `a`，最好将其名称变成 `double!` 形式，能有很好的提示效果。这也是贯穿了 Julia 标准库的惯例。

- ❑ 不定义 `x->f(x)` 这样的匿名函数。高阶 (higher-order) 函数通常会使用匿名函数进行调用，但如果某函数可直接传参，就不要用匿名函数进行封装，应使用 `map(f, a)` 代替 `map(x->f(x), a)`。
- ❑ 条件表达式不需要圆括号，例如应写 `if a == b`，而不是 `if (a == b)`。
- ❑ 不要过度使用异常处理逻辑，更好的选择是充分调试避免潜在的错误。

编程实战

决策树是机器学习领域经典的算法，不仅能够用于分类问题，也能够用于回归问题，已经在各行业的数据挖掘中得到了广泛的应用，而且出现了大量的衍生算法，包括随机森林、提升树等。如果读者有进入机器学习领域的想法，或者已经进入了这个充满挑战的领域，决策树类的模型算法都是绕不开的、必须要掌握的基本方法。

本章将选择决策树作为实战项目，这也是笔者在读者学习 Julia 语言后实践过的项目。

从零开始地实现一套算法模型，是掌握熟知该算法的最好方式，是远比在现有算法包上调调参数更为有效的学习方式。与此同时，在模型实现的过程中，也能够熟悉掌握一门编程语言的各种技巧，因为你需要想尽各种办法去实现模型的复杂逻辑，去提升模型的整体性能，甚至还要考虑模型的分布式部署与并行化问题。所有这些实际问题，都促使我们不断深入地了解一门编程语言，不断提高编程水平。

17.1 决策树基本概念

在使用 Julia 语言实现决策树之前，显然我们需要先熟悉决策树的各种理论与核心算法。因为超出本书范围，所以需要读者自行学习。不过，为了阐述方便，下面以一个实际生活中的例子说明决策树的基本思想。

侄女年龄已经二十五六岁了，好事的姑姑比谁都着急，经常给侄女介绍各种对象，这次又有个自觉很不错的候选人，便急忙和侄女说了起来。两人的对话如下：

姑姑：有个小伙子很不错，要不要见见？

侄女：没超过 30 岁吧？

姑姑：没有没有，28 岁。

侄女：是不是又比我还矮？

姑姑：怎么会呢。这个小伙有一米七八高呢。

侄女：那还好。有房吗？

姑姑：放心，既有车又有房，绝对靠谱！

侄女：那挺好的，在哪里上班？是 IT 男吗？

姑姑：自然是了，姑姑可都是按照你的要求找的。

侄女：哦，那挺好的，可以考虑下。那约个时间见一下吧。

从上述的对话中可见，侄女对男朋友的选择有着不少门槛，见与不见都要看候选人是否满足各项条件，尤其还有个必须是 IT 男的古怪癖好。暂不论这个 IT 男是否会看上这个姑娘，我们只知道，只要有一条不符合，姑娘便会选择不见候选人。

我们可以把这个过程稍微地转换一下，用一种形式化的方式进行理解。对于一个候选人来说，侄女的决策结果只会是“见”或“不见”，而且这两个结果是互斥的，非此即彼。每个候选人在年龄、身高、房产、IT 男四个方面都逐项满足，才会得到这个姑娘的青睐。这样的决策过程并不复杂，每个条件都存在两个选择分支，我们可以将这个判断的过程形象地表达出来，如图 17-1 所示，这是二叉树结构。

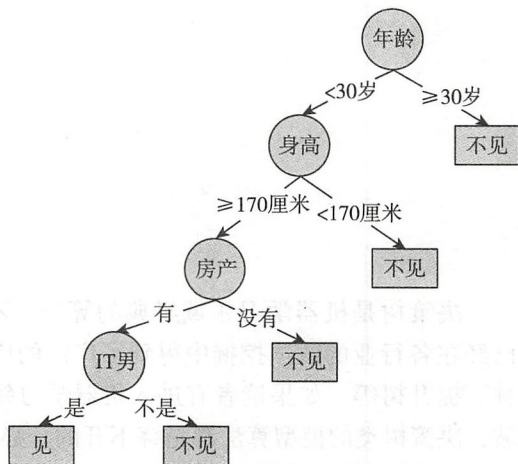


图 17-1 找对象二叉树决策流程

这样的一棵二叉树描述了我们日常生活中对很多事情的决策流程，也正是机器学习领域中决策树方法的思想起源。

在上述的对话中，能够猜测出，姑姑给侄女介绍了不少对象，但侄女都不太满意。之所以如此，多半是因为姑姑一开始并不清楚侄女的选择标准，而傲娇的侄女估计一开始也没有明说。不过，多次的推荐不果，姑姑根据之前的情况多少能够推断出了侄女的喜好，才有了这次的顺利；亦或侄女实在不耐烦姑姑胡乱地介绍，终于明确告知了姑姑自己的标准。

假设，在侄女没有明确告知择偶标准之前，姑姑已经介绍了 9 个对象，每个候选人的情况分别如表 17-1 所示。其中有房产的用 1 表示，没有房产的用 0 表示；是 IT 行业的用 1 表示，不是的则用 0 表示；而约见了的同样用 1 表示，而没见的用了 0 表示。

表 17-1 男朋友候选人条件及被接见的结果

候选人编号	年龄	身高	房产	职业	见没见	候选人编号	年龄	身高	房产	职业	见没见
1	31	168	0	1	0	6	26	166	1	1	1
2	35	171	1	1	0	7	28	172	1	1	1
3	32	173	1	0	0	8	28	161	0	1	0
4	30	165	0	1	0	9	25	173	1	0	0
5	24	171	0	1	0						

从表中能够看出，在之前姑姑推荐的候选人中，幸运儿并不多，原因自然是姑姑对侄女心中已经制定的标准无从知晓。那么，是否能够根据表中列出的数据情况，找到一个方法帮助姑姑推断出侄女的择偶标准，省得姑姑老是白忙活？

这种从属性数据及目标结果标签（见没见）对样本进行分类（适不适合推荐给侄女）的过程，便是对决策树模型进行构建的过程，即基于训练数据的学习（Training 或 Learning）过程。而如果一旦决策树模型训练完成，将任意一个候选人资料输入模型，推断是否被约见的过程便是预测（Prediction）过程。

17.2 决策树分类器的实现

当然，无须从零开发一套决策树算法，在 Julia 社区的第三方库中有个 `DecisionTree.jl` 库，已经非常好地实现了决策树相关的算法。通过对该库实现过程的详细介绍，我们也能够对 Julia 的实践过程有个不错的了解。不过，对于读者来说，笔者仍然建议从零开始，尝试自己实现决策树算法，然后与 `DecisionTree.jl` 库进行比对。这个过程无论是对决策树的学习还是对 Julia 语言的学习，都会有极大的好处。

`DecisionTree.jl` 库不但支持基本的决策树模型，同时还支持随机森林、提升树模型，而且提供了包括分类和回归两种工作模式。在调用方式上，该库在操作界面方面采用了与 Scikit-Learn 兼容的接口设计，使得熟悉这个影响力极大的机器学习库的用户能够更快地使用 `DecisionTree.jl` 库。笔者已经对该库的主要代码进行了详细的注释，读者可以通过本书提供的网址 <https://gitee.com/juliapro/DecisionTree.jl.git> 获得已经注释过的源代码。

1. `DecisionTree.jl` 基本使用

在研究 `DecisionTree.jl` 库之前，我们先看一下已经实现的决策树模型是如何使用的。

首先，通过 `Pkg.add("DecisionTree")` 命令在本机上安装该库。然后，在代码中执行语句 `using DecisionTree` 便可使用该库中的分类与回归模型，如 `DecisionTreeClassifier`、`RandomForestClassifier`、`AdaBoostStumpClassifier`、`DecisionTreeRegressor`、`RandomForestRegressor` 等。

我们采用经典的鸢尾花（Iris）数据作为样本，以分类模型为例，简单介绍该库的使用。主要的代码如下：

```
using RDatasets: dataset
using DecisionTree

# 导入样本数据，并分离特征与标签列
iris = dataset("datasets", "iris")
features = convert(Array, iris[:, 1:4]);
labels = convert(Array, iris[:, 5]);

# 创建分类器对象，限定最大深度为2
model = DecisionTreeClassifier(max_depth=2)
```



```
# 基于样本, 对分类器模型进行训练
fit!(model, features, labels)

# 打印训练好的模型树结构
print_tree(model.root, 5)
```

其中的 `Rdatasets` 库同样是 Julia 的第三方库, 可通过 R 语言接口方便地使用一些经典的数据集, 当然, 在使用该库之前需要使用 `Pkg.add()` 命令安装好它。

在语句 `iris = dataset("datasets", "iris")` 执行时, 会打印出样本数据, 如下所示:

150×5 DataFrames.DataFrame						
Row	SepalLength	SepalWidth	PetalLength	PetalWidth	Species	
1	5.1	3.5	1.4	0.2	setosa	
2	4.9	3.0	1.4	0.2	setosa	
3	4.7	3.2	1.3	0.2	setosa	
4	4.6	3.1	1.5	0.2	setosa	
5	5.0	3.6	1.4	0.2	setosa	
⋮						
145	6.7	3.3	5.7	2.5	virginica	
146	6.7	3.0	5.2	2.3	virginica	
147	6.3	2.5	5.0	1.9	virginica	
148	6.5	3.0	5.2	2.0	virginica	
149	6.2	3.4	5.4	2.3	virginica	
150	5.9	3.0	5.1	1.8	virginica	

样本总计有 150 行, 有 4 个特征, 并有 `setosa`、`versicolor` 和 `virginica` 三类。在将样本集分离出特征数据与标签数据后, 执行 `fit!(model, features, labels)` 便可使用分类器对象 `model` 对样本进行拟合, 实现决策树的训练。

在辅助函数 `print_tree()` 调用后, 便能够对训练出的决策树结构进行查看, 如下所示:

```
Feature 3, Threshold 2.45
L-> setosa : 50/50
R-> Feature 4, Threshold 1.75
  L-> versicolor : 49/54
  R-> virginica : 45/46
```

意思是: 根节点以特征 3 为基础分裂, 左右分支的分裂阈值为 2.45, 左分支有 50 个样本, 而且都属于 `setosa` 类; 右分支又基于阈值 1.75 对特征 4 进行再次分裂, 其左分支共 54 个样本, 其中有 49 个为 `versicolor` 类, 其右分支共 46 个样本, 有 45 个为 `virginica` 类。直观来看, 模型最终的分类效果还是可以的, 每个分支中的类别纯度都比较高。

至此, 模型的第一阶段训练工作便完成了, 第二阶段便是对某个样本数据进行分类。例如某个鸢尾花的特征向量为 `[5.9, 3.0, 5.1, 1.9]`, 调用 `predict()` 函数便可获得它的类别, 如下:

```
predict(model, [5.9, 3.0, 5.1, 1.9])
```

会输出如下结果:

```
CategoricalArrays.CategoricalString{UInt8} "virginica"
```

可见该样本的类别为 `virginica` (结果中的前缀内容仅为模型的附加信息, 可以忽略)。如果要查看分类时对各类的预测概率, 可执行以下命令:

```
predict_proba(model, [5.9, 3.0, 5.1, 1.9])
```

执行后会得到如下结果:

```
3-element Array{Float64,1}:
 0.0
 0.0217391
 0.978261
```

即该样本属于 `setosa`、`versicolor` 和 `virginica` 三类的概率分别为 0.0、0.0217391 和 0.978261, 所以确定类别为 `virginica`。

2. Scikit-learn 接口

Scikit-learn 项目最早由数据科学家 David Cournapeau 在 2007 年发起, 是 Python 语言中针对机器学习应用而发展起来的开源框架, 其基本功能主要分为六部分: 分类、回归、聚类、数据降维、模型选择和数据预处理, 其中的分类算法又包括支持向量机 (SVM)、最近邻算法、逻辑回归、决策树、随机森林以及多层感知器 (MLP) 神经网络等。

该项目已经被研究人员及开发者广泛地使用。为了能够让人们更快、更容易地使用 Julia 的相关库, 聪明的 Julia 开发者专门建立了 `ScikitLearn.jl` 项目及 `ScikitLearnBase.jl` 项目, 使得这种跨语言的迁移式学习更为简单。

在我们对 `DecisionTree.jl` 的使用中, 无论是模型的名称还是函数等的调用, 都与项目 Scikit-learn 中的表达方式极为类似。所以这里我们有必要简单介绍一下统一了封装接口的项目 `ScikitLearnBase.jl`。读者可以通过 <https://gitee.com/juliapro/ScikitLearnBase.jl.git> 地址获得该项目的源码, 笔者也对其主要内容进行了翻译, 以便于读者更好的理解。

事实上, 该项目极为简单, 仅有一个源代码脚本, 而且行数也不多。所有的宏、函数等都在脚本 `src/ScikitLearnBase.jl` 中一个名为 `ScikitLearnBase` 的模块中定义。

该脚本的头几行代码为:

```
__precompile__()
module ScikitLearnBase
using Compat
```

其中第一行的 `__precompile__()` 用于显示地告知 Julia 开发环境, 在第一次引入该包时需对该脚本进行预编译; 第二行则是模块定义的开始; 第三行使用 `using` 关键字引入了该包唯一的依赖库 `Compat`。



提示 `Compat.jl` 库是 Julia 官方维护的开发辅助工具包, 专门用于处理 Julia 语言不同版本之间的兼容性。通过使用该包, 当 Julia 版本更新时, 各第三方库中已经开发的功能能够自动继承最新版本的特性, 而无须包的作者对已有的代码做太多的变更, 这是一个非常聪明的为语言提供向后兼容能力的做法。

下面是一个看起来较为复杂的宏, 如下所示:

```
macro declare_api(api_functions...)
    esc(:begin
        $([Expr(:function, f) for f in api_functions]...)
        # Expr(:export, f) necessary in Julia 0.3
        $([Expr(:export, f) for f in api_functions]...)
        const api = [$([Expr(:quote, x) for x in api_functions]...)]
    end))
end
```

该宏采用了可变参数，能够接受任意个参数。实现中的 `esc()` 是一种专门在 `macro` 中使用的函数，用于阻止 `Expr` 类型中将嵌入的变量自动转为符号变量。其中的 `begin` 与 `end` 结构将内部的多个表达式组合成了复合表达式。该宏只是 `ScikitLearnBase` 将名字导出的辅助工具，关于其具体内容有兴趣读者可以深入研究下。

在导出各种名字后，便声明了三个抽象类型，对应于 `Scikit-Learn` 中的同名结构：

```
@compat abstract type BaseEstimator end
@compat abstract type BaseClassifier <: BaseEstimator end
@compat abstract type BaseRegressor <: BaseEstimator end
```

从中可见，`BaseEstimator` 是顶级抽象类型，而 `BaseClassifier` 和 `BaseRegressor` 都是其子类型，分别对应于分类器或回归器模型的接口。可以说，抽象类型的声明是对接口进行统一的首选方式。

在此之后，便是各种常规方式定义的函数与宏。有兴趣的读者可以对照 `Scikit-Learn` 的代码研究一下，这里不再多做介绍。

3. 模型调用接口

下面仍回到对 `DecisionTree.jl` 的介绍中，详细讲解其对决策树类模型的实现过程。

在 `DecisionTree.jl` 的源码目录 `src` 中，有一个 `scikitlearnAPI.jl` 脚本，专门定义了适配于 `Scikit-Learn` 的决策树类型与函数命名方式。

该脚本第一行的语句为：

```
import ScikitLearnBase: BaseClassifier, BaseRegressor, predict, predict_proba,
    fit!, get_classes, @declare_hyperparameters
```

通过引入前述的 `ScikitLearnBase` 包，获得与 `Scikit-Learn` 保持一致的接口名称。至于其中的宏 `@declare_hyperparameters` 的意义，可以参考 `ScikitLearnBase.jl` 包中的 `docs/API.md` 文件，笔者已经对其进行了翻译，读者可以通过它作进一步的了解。

接下来，定义了决策树分类器的基本结构，如下所示：

```
"""
    DecisionTreeClassifier(; pruning_purity_threshold=1.0,
                           max_depth::Int=-1,
                           min_samples_leaf::Int=1,
                           min_samples_split::Int=2,
                           min_purity_increased::Float=0.0,
                           n_subfeatures::Int=0,
                           rng=Base.GLOBAL_RNG)
    Decision tree classifier. See [DecisionTree.jl's documentation](https://github.com/bensadeghi/DecisionTree.jl)
```


Hyperparameters:

- `pruning_purity_threshold`: (post-pruning) merge leaves having `>=thresh` combined purity (default: no pruning)
- `max_depth`: maximum depth of the decision tree (default: no maximum)
- `min_samples_leaf`: the minimum number of samples each leaf needs to have (default: 1)
- `min_samples_split`: the minimum number of samples in needed for a split (default: 2)
- `min_purity_increase`: minimum purity needed for a split (default: 0.0)
- `n_subfeatures`: number of features to select at random (default: keep all)
- `rng`: the random number generator to use. Can be an `Int`, which will be used to seed and create a new random number generator.

Implements `fit!`, `predict`, `predict_proba`, `get_classes`

"""

mutable struct DecisionTreeClassifier <: BaseClassifier # 将分类器声明抽象类型Base-Estimator的子类型

pruning_purity_threshold::Float64 # no pruning if 1.0

max_depth::Int

min_samples_leaf::Int

min_samples_split::Int

min_purity_increase::Float64

n_subfeatures::Int

rng::AbstractRNG

root::Union{LeafOrNode, Nothing}

classes::Union{Vector, Nothing}

内部构造函数, 键值参数均提供了默认值, 使用new创建对象

DecisionTreeClassifier(;pruning_purity_threshold=1.0, max_depth=-1,
min_samples_leaf=1, min_samples_split=2,
min_purity_increase=0.0, n_subfeatures=0,
rng=Base.GLOBAL_RNG,
root=nothing, classes=nothing) =

new(pruning_purity_threshold,
max_depth, min_samples_leaf, min_samples_split,
min_purity_increase, n_subfeatures, mk_rng(rng), root, classes)

end

上述的内容分两个部分, 一个是可变复合类型 DecisionTreeClassifier 的定义, 另一部分是对该类型的详细注释。

类型 DecisionTreeClassifier 中的成员除了控制模型的各种超级参数 (Hyperparameters) 外, 也有用于记录训练结果的树型结构变量 root 和存储标签状态的 classes 成员, 而且这两个变量的类型都采用了 Union 的方式进行限定, 表示可以无效也可以有效 (只是类型不同)。

另外, 在类型的内部, 提供了一个内部构造方法用于创建该类型的对象, 参数均以键值的方式提供, 并均提供了默认值。所以一般情况下, 用户在使用该分类器类型时, 可以不用显式地提供任何参数, 便可创建一个分类器对象。至于这些参数的意义, 读者可以在深刻理解决策树的原理后, 自行查阅 Scikit-Learn 相关的资料去了解。

紧随该类型定义, 定义了一个辅助函数:

```
get_classes(dt::DecisionTreeClassifier) = dt.classes
```


用于获得分类器模型中记录的类别数组。例如鸢尾花的例子，调用该函数会返回如下内容：

```
3-element Array{CategoricalArrays.CategoricalString{UInt8},1}:
 "setosa"
 "versicolor"
 "virginica"
```

此后，该脚本中便定义了针对该分类器的拟合（训练）函数，如下所示：

```
# 训练函数，基于样本X与目标值（标签）y构建决策树分类器DecisionTreeClassifier
function fit!(dt::DecisionTreeClassifier, X, y)
    dt.root = build_tree(y, X, dt.n_subfeatures, dt.max_depth, dt.min_samples_leaf,
                        dt.min_samples_split, dt.min_purity_increase; rng=dt.rng)
    if dt.pruning_purity_threshold < 1.0
        dt.root = prune_tree(dt.root, dt.pruning_purity_threshold)
    end
    dt.classes = sort(unique(y))
    dt
end
```

该函数是对内部实现细节的一层封装，所以看起来非常简洁。



注意 由于其在执行过程中会对 DecisionTreeClassifier 类型的参数 dt 做出修改，所以函数在命名时附加了感叹号标识，用于提醒开发者的注意。

接着，提供的接口函数便是对训练好的模型进行应用，即预测函数，如下所示：

```
predict(dt::DecisionTreeClassifier, X) = apply_tree(dt.root, X)

predict_proba(dt::DecisionTreeClassifier, X) = apply_tree_proba(dt.root, X, dt.classes)
```

这两个函数前例已经尝试使用过，并不复杂；在实现上，这两个函数也是对内部具体实现函数 apply_tree() 的一层封装。

在这两个函数之后，提供了一个辅助函数，用于打印模型对象 model 的内部字段信息：

```
function show(io::IO, dt::DecisionTreeClassifier)
    println(io, "DecisionTreeClassifier")
    println(io, "max_depth:           $(dt.max_depth)")
    println(io, "min_samples_leaf:           $(dt.min_samples_leaf)")
    println(io, "min_samples_split:          $(dt.min_samples_split)")
    println(io, "min_purity_increase:        $(dt.min_purity_increase)")
    println(io, "pruning_purity_threshold:    $(dt.pruning_purity_threshold)")
    println(io, "n_subfeatures:              $(dt.n_subfeatures)")
    println(io, "classes:                    $(dt.classes)")
    println(io, "root:                       $(dt.root)")
end
```

看起来很简洁，输出的介质限定了 IO 类型，即可以为控制台或文件等流对象。在输出时，dt 成员内容的获取采用了字符串变量展开的方式。虽然这未必是最好的方式，但在性能不敏感时，对于简洁清晰的代码表述，也是一个比较好的选择。

对于上文中鸢尾花数据训练得出的模型，如果调用该函数，会显示类似下面的内容：

```
DecisionTreeClassifier
```

```

max_depth:      2
min_samples_leaf: 1
min_samples_split: 2
min_purity_increase: 0.0
pruning_purity_threshold: 1.0
n_subfeatures: 0
classes:        CategoricalArrays.CategoricalString{UInt8}["setosa",
"versicolor", "virginica"]
root:           Decision Tree
Leaves:         3
Depth:          2

```

后面的代码是其他模型的定义，包括随机森林、提升树和回归模型等。实现的语句基本与 `DecisionTreeClassifier` 的实现类似，这里不再多做介绍。

4. 树型结构表达

在二叉树结构中，节点主要分为两类，一种是有孩子的，另一种无孩子的（也称为叶子）。同时，对于有孩子的节点，无论是子节点还是父节点都有着相似的结构。故此，在用树型进行表达时，需要分别为有孩子的节点及叶子两种情况定义不同的类型，而在有孩子的节点内部还需要包括左右孩子的节点对象。

树型结构的类型与相关操作函数主要在脚本 `src/DecisionTree.jl` 中实现，代码如下所示：

```

# 不可变复合类型，叶子
struct Leaf
    majority::Any
    values::Vector
end

# 可变复合类型，节点
mutable struct Node
    featid::Integer
    featval::Any
    left::Union{Leaf,Node} # 分支节点：可以是叶子或普通节点
    right::Union{Leaf,Node} # 分支节点：可以是叶子或普通节点
end

```

在实现中，叶子类型 `Leaf` 被定义为不可变复合类型，即一旦创建其内部字段便不可更改；而普通节点类型 `Node` 被定义为可变复合类型，而且内部字段完全不同：除了两个属性字段，还有 `left` 及 `right` 两个成员，记录当前节点所属的孩子节点。而且，由于孩子可能是普通节点，也可能是叶子节点，所以采用了 `Union{}` 类型对这两个成员字段进行了类型限定。另外，为了使用方便，将两类节点的类型联合声明为一个新的类型，名称为 `LeafOrNode`，代码如下：

```
const LeafOrNode = Union{Leaf,Node}
```

在决策树后续的计算过程中，有时候需要将 `Leaf` 类型作为 `Node` 类型处理，为此，通过以下语句对 `convert()` 函数扩展，能够将 `Leaf` 类型转为 `Node` 类型：

```
convert{::Type{Node}, x::Leaf} = Node(0, nothing, x, Leaf(nothing, [nothing]))
```

而且因为后续会涉及两个类型的混合操作，又为这两个类型定义了提升规则：

```
promote_rule(::Type{Node}, ::Type{Leaf}) = Node
promote_rule(::Type{Leaf}, ::Type{Node}) = Node
```

即在二元操作中，左操作数与右操作数分为 Node 或 Leaf 时，不论左右的顺序，均会被提升为 Node 类型进行处理。

在接下来的实现代码中，提供了对这两个基础类型的操作函数，而且对于 Node 或 Leaf 这两种不同的类型，提供了同名的操作接口：

```
length(leaf::Leaf) = 1
length(tree::Node) = length(tree.left) + length(tree.right)
```

这两个函数分别用于获得普通节点及叶子节点的长度（孩子数量），因为同名，所以在调用时会触发 Julia 的多态分发机制。函数 length() 会根据参数类型自动选择对应的具体实现方法。而且，在第二个方法中，存在递归调用，仅在 tree.left 或 tree.right 为 Leaf 类型时才会结束递归。

类似地，可以通过 depth() 函数求取节点的深度。该函数同样存在两个方法，分别用于 Leaf 类型与 Node 类型。而且对于 Node 也属于递归调用，直到 tree.left 或 tree.right 均为叶子时才结束递归。具体代码如下：

```
depth(leaf::Leaf) = 0
depth(tree::Node) = 1 + max(depth(tree.left), depth(tree.right))
```



提示 有兴趣的读者可以对比一下这两个函数的区别，也可以思考下为何其中的第二个 depth() 方法需要有加 1 的操作。

此后的 print_tree() 函数是辅助工具，用于打印树的内部结构与属性信息，实现代码如下：

```
function print_tree(leaf::Leaf, depth=-1, indent=0)
    matches = find(leaf.values .== leaf.majority)
    ratio = string(length(matches)) * "/" * string(length(leaf.values))
    println("$(leaf.majority) : $(ratio)")
end

function print_tree(tree::Node, depth=-1, indent=0)
    if depth == indent
        println()
        return
    end
    println("Feature $(tree.feaid), Threshold $(tree.feaval)")
    print("    " ^ indent * "L-> ")
    print_tree(tree.left, depth, indent + 1)
    print("    " ^ indent * "R-> ")
    print_tree(tree.right, depth, indent + 1)
end
```

这两个方法分别针对 Leaf 类型与 Node 类型，同样需要多态分发机制的支持。第二个方法中同样存在递归行为，会在 tree.left 或 tree.right 为 Node 时反复调用第二个方

法；再逐层递归，直到两个成员为 Leaf 类型时，才会自动调用第一个参数类型为 Leaf 的实现方法，结束递归。

5. 效果评价方法

包括决策树在内的各种机器学习方法，都是涉及一个基本问题：模型对训练数据的拟合效果到底如何？模型对新数据的预测能力如何？

这两个问题也是算法人员工作的核心，即在不断提升模型效果指标的同时，还需要确保模型在测试集（未参与模型训练的数据）上表现良好，具有很强的泛化能力。

包 `DecisionTree.jl` 在 `src/measures.jl` 中实现了专门的量化评价指标。首先，看一个复合类型的定义：

```
struct ConfusionMatrix
    classes::Vector          # 标签（类别）向量
    matrix::Matrix{Int}      # 模型对样本分类时，识别到各类的数量
    accuracy::Float64        # 准确度
    kappa::Float64
end
```

其中以不可变复合类型定义了混淆矩阵（Confusion Matrix）结构。

混淆矩阵是一种针对分类问题的评价方法，能够在 TP（正确的肯定）、FN（错误的否定，漏报）、FP（错误的肯定，误报）以及 TN（正确的否定）四个方面进行详细的评估。这种评价方式能比单纯的精度 / 准确度指标更为详尽地反映模型的“好坏”，不会在正负样本数量不均衡的情况下，出现误导的评价。

该脚本中的 `confusion_matrix()` 函数用于根据预测的测试结果生成有效的混淆矩阵对象，具体的实现代码这里不再给出，有兴趣的读者可以在本书提供的源码库中查看。

对于回归问题，因为与分类问题有着很大的区别，所以评价方法也有所不同。包中采用的方法是简单的均平方误差（MSE），实现的函数为：

```
function mean_squared_error(actual, predicted)
    @assert length(actual) == length(predicted)
    return mean((actual - predicted).^2)
end
```

其中的参数 `actual` 是实际的 y 值（可确认的正确值），而 `predicted` 为模型预测的 y 值。由于需要两者对比，所以代码的第一条语句便是要求两者的长度一致，通过 `@assert` 宏对该条件进行了验证。然后，便是逐元地计算数组 `actual` 与 `predicted` 之间的误差。代码中采用了点操作符进行矢量化计算，在对结果误差数据逐元计算平方值后，调用 `mean()` 函数求取误差平方序列的均值。

另外，分别为分类器与回归器提供了交叉验证（Cross Validation）方法，函数名均有 `nfoldcv_` 前缀。而且，针对决策树、随机森林和提升树都实现了这个接口，不过调用的都是 `_nfoldcv()` 函数，只是通过不同的函数来区分不同模型的交叉验证。这是一种将核心实现进行集中封装的方式，更利于减少代码量，提高可维护性。

6. 信息熵计算

熵是决策树中最基础的概念，不但用于每次分裂时最优特征的选择，也用于确定样本分拆时特征的最佳分裂阈值点。在 `src/util.jl` 脚本中，熵的实现代码如下：

```
# n为总样本数
# ns 为其中各类别标签样本数量
@inline function entropy(ns::Array{Int64}, n::Int64)
    s = 0.0 # 临时变量，记录熵计算时的中间值
    @simd for k in ns
        if k > 0 # 类别数>0才有意义
            s += k * log(k) # 计算各类别的信息熵，不过暂时不以n作为除数算法概率
        end
    end
    return log(n) - s / n # 统一除以n，折算出最终的熵
end
```

该函数接受两个参数，分别是各类别标签对应的样本数 `ns` 及总样本数 `n`；并限定了两者的具体类型，分别为元素类型为 `Int64` 的数组和 `Int64` 类型的标量。

在具体实现时，变量 `s` 作为临时的局部变量，记录着熵计算的中间值；然后以 `for` 结构遍历所有的标签统计值，对熵的中间值进行累加。因为不同标签统计量对熵的计算无顺序依赖，所以实现中对 `for` 结构使用了宏 `@simd` 进行修饰，使得累加能够并行处理。

在该段代码中，函数 `entropy()` 用 `@inline` 宏所修饰，用于告知编译器该函数需要进行内联处理。所谓内联，指的是编译器会将内联函数的实现直接嵌入到被调用处，在执行时不用再反复执行函数的调用过程，省去了函数调用的消耗，能够在一定程度上提升性能和效率。不过，这种方式会增大执行文件的体积，幸运的是，这种增大很多时候并不会有什么大的影响。



提示 在 Julia 中，对于实现短小的函数，编译器会自动处理为内联函数；对于较大的函数，如果要进行内联，则需显式地使用 `@inline` 宏对该其进行标识，明确地告知编译器需对该函数进行内联处理。当然，可以使用 `@noinline` 对该小函数进行标识，以阻止编译器自动内联。

在 `DecisionTree.jl` 的代码中，函数 `entropy()` 是需要被频繁调用的，所以该库将之进行内联处理，是一个极为明智的处置。

7. 训练过程实现

有了之前的各种基础后，我们可以实现决策树训练的主要流程了。在训练过程中，对树进行构建的主要函数是 `build_tree()`，在脚本 `src/classification/main.jl` 中实现的代码如下：

```
# 对决策树进行训练，一般只需提供样板的标签值及特征序列就行
function build_tree(labels::Vector, features::Matrix, n_subfeatures=0, max_depth=-1,
    min_samples_leaf=1, min_samples_split=2,
    min_purity_increase=0.0; rng=Base.GLOBAL_RNG)
```

```

rng = mk_rng(rng)::AbstractRNG

# 未提供有效默认参数的, 转为有效参数
if max_depth < -1
    error("Unexpected value for max_depth: $(max_depth) (expected: max_depth >= 0,
        or max_depth = -1 for infinite depth)")
end
if max_depth == -1
    max_depth = typemax{Int64}()
end
if n_subfeatures == 0
    n_subfeatures = size(features, 2)
end
min_samples_leaf = Int64(min_samples_leaf)
min_samples_split = Int64(min_samples_split)
min_purity_increase = Float64(min_purity_increase)

# 实际构造函数
t = treeclassifier.fit(features, labels, n_subfeatures, max_depth,
    min_samples_leaf, min_samples_split,
    min_purity_increase, rng=rng)

# 内部嵌套定义的函数
function _convert(node :: treeclassifier.NodeMeta, labels :: Array)
    if node.is_leaf
        distribution = []
        for i in 1:length(node.labels)
            counts = node.labels[i]
            for _ in 1:counts
                push!(distribution, labels[i])
            end
        end
        return Leaf(labels[node.label], distribution)
    else
        left = _convert(node.l, labels)
        right = _convert(node.r, labels)
        return Node(node.feature, node.threshold, left, right)
    end
end
return _convert(t.root, t.list)
end

```

该函数需要用户输入的两个参数是 `Vector` 类型的 `labels`, 代表样本的标签序列; 还有就是 `Matrix` 类型的 `features`, 代表样本特征向量的集合。

其他的参数主要用于对模型的构建优化过程进行控制, 包括最大深度、叶子节点分配的最少样本数、最小熵纯度等, 不过包已经提供了默认值, 可以不提供这些参数。当然, 这些参数默认值并非是实际使用的值, 在调用模块 `treeclassifier` 中的 `fit()` 函数执行实际的训练构建前, 这些参数会根据样本情况提供有效的取值。在实际的应用中, 为了最佳性能, 这些控制模型的超级参数往往都需要优化、调整。

在仔细查看 `treeclassifier.fit()` 函数的内部实现之前, 继续查看 `build_tree()` 函数后续的实现内容, 会发现其中有一个 `_convert()` 函数的定义。这是一个在 `build_`

tree() 内部定义的嵌套函数，用于将 fit 训练出的 NodeMeta 树型结构中的各种信息进行进一步处理，从而将其信息转移到 Node 和 Leaf 类型构成的树型结构中。

其中的 NodeMeta 类型定义在 src/classification/tree.jl 脚本的 treeclassifier 模块中，是另一套树型结构，用于辅助决策树的分裂，具体实现如下：

```
mutable struct NodeMeta
    l          :: NodeMeta          # 左孩子
    r          :: NodeMeta          # 右孩子
    labels     :: Array{Int64}
    label      :: Int64              # 本节点概率最高的分类标签
    feature    :: Int64              # 在本节点作为分裂依据的特征编号（列索引）
    threshold  :: Any                # 分裂阈值
    is_leaf    :: Bool

    depth      :: Int64              # 本节点深度
    region     :: UnitRange{Int64}
    features   :: Array{Int64}

    split_at   :: Int64
    NodeMeta(features, region, depth) = (
        node = new();
        node.depth = depth;
        node.region = region;
        node.features = features;
        node.is_leaf = false;
        node)
end

mutable struct Tree{T}
    root :: NodeMeta
    list :: Array{T}
end
```

该类型为可变复合类型，其作用正如其名，用于记录节点在构建决策树过程中所需的各种必要信息，包括当前节点概率最高的所属标签以及该节点在分裂时所依据的特征，同时也记录了分裂阈值和位置、样本 Slice 等信息。

另外，一个名为 Tree 的参数化可变复合类型，是对 NodeMeta 的再次封装，其中的 list 记录的是标签列表。

接下来便是 fit() 的实现了，具体实现的代码如下：

```
function fit(X          :: Matrix{T},
            Y          :: Vector,
            max_features :: Int64,
            max_depth   :: Int64,
            min_samples_leaf :: Int64,
            min_samples_split :: Int64,
            min_purity_increase :: Float64;
            rng=Base.GLOBAL_RNG :: AbstractRNG) where T <: Any

    # X是样本特征，一行一个样本，列为特征
    # 取得样本数及特征数
    n_samples, n_features = size(X)
```

```

# 对样本进行编码
label_list, _Y = assign(Y)
# 不同的标签数
n_classes = Int64(length(label_list))

# 检查输入参数的合法性
check_input(
    X, _Y,
    max_features,
    max_depth,
    min_samples_leaf,
    min_samples_split,
    min_purity_increase)

# 生成元素依次为1: 样本数的数组序列编号
indX = collect(Int64(1):Int64(n_samples))

# 对决策树进行初始化
tree = let
    # 树的根节点: 对所有特征进行处置, 所有样本作为region, 节点深度为0
    @inbounds root = NodeMeta(collect(1:n_features), 1:n_samples, 0)
    # 创建决策树Tree的对象: 根节点对象, 标签序列
    Tree(root, label_list)
end

stack = NodeMeta[ tree.root ]

nc = Array{Int64}(n_classes)
ncl = Array{Int64}(n_classes) # 记录左分支中各类别的数量
ncr = Array{Int64}(n_classes) # 记录右分支中各类别的数量
Xf = Array{T}(n_samples)
Yf = Array{Int64}(n_samples)

@inbounds while length(stack) > 0
    node = pop!(stack)
    _split!(
        X,
        _Y,
        node,
        n_classes,
        max_features,
        max_depth,
        min_samples_leaf,
        min_samples_split,
        min_purity_increase,
        indX,
        nc, ncl, ncr, Xf, Yf,
        rng)
    if !node.is_leaf
        fork!(node)
        push!(stack, node.r)
        push!(stack, node.l)
    end
end
end
return tree

```



```
end
end
```

这是一个参数化函数，其中的类参 `T` 用于适应不同元素类型的特征矩阵（特征向量集合）。

对于不同类型的标签，该函数首先采用 `assign()` 函数对其进行重新编码，无论标签是浮点型、整型、字符串还是其他类型，都会转为整型编码。这里详细讲一下这个编码函数的实现过程，代码如下：

```
function assign(Y :: Array{T}) where T<:Any

    # 收集样本出现的各种可能标签
    # 并利用集合特性对样本集的Y进行去重，并放在元素唯一性的集合中，为区分称之为类别
    label_set = Set{T}()
    for y in Y
        push!(label_set, y)
    end

    # 将得到的标签集合转换到数组结构
    label_list = collect(label_set)

    # 利用字典结构，为标签进行编码，而码值便是标签在数组label_list中的下标（1-Based）
    label_dict = Dict{T, Int64}()

    @inbounds for i in 1:length(label_list)
        label_dict[label_list[i]] = i    # 将label_list[i]对应的标签码值记录为对应的下标值i
    end

    # 按原样本集的顺序，生成新的样本标签序列，但标签值均依序被替换为整型编码值
    _Y = Array{Int64}(length(Y))
    @inbounds for i in 1:length(Y)
        _Y[i] = label_dict[Y[i]]
    end

    return label_list, _Y                # 同时返回有序不重复的样本序列，及样本标签码序列
end
```

该函数同样为参数化方法，类参 `T` 用于适应 `Any` 类型的标签。在实现中，首先利用 `Set` 的特性对样本标签进行去重处理。然后，使用 `collect()` 函数将标签集合转为数组序列。基于此序列，以 `Dict` 类型为基础，建立标签内容与其在标签序列 `label_list` 中索引下标值的对应关系。`Dict` 结构的好处在于，在编码后能更方便地解码。



提示 在处理数组遍历的过程中，代码中使用了 `@inbounds` 宏，能够避免 Julia 编译器自动进行数组索引范围检查，有助于提高性能。

在对样本标签序列进行编码后，下一步便是使用 `check_input()` 对输入参数的有效性进行检查。该函数的具体实现如下：

```
function check_input(X :: Matrix,
                    Y :: Array{Int64, 1},
                    max_features :: Int64,
                    max_depth :: Int64,
                    min_samples_leaf :: Int64,
```

```

        min_samples_split    :: Int64,
        min_purity_increase :: Float64)
n_samples, n_features = size(X)
if length(Y) != n_samples
    throw("dimension mismatch between X and Y ($(size(X)) vs $(size(Y)))")

elseif n_features < max_features
    throw("number of features $(n_features) "
          * "is less than the number of "
          * "max features $(max_features)")

elseif min_samples_leaf < 1
    throw("min_samples_leaf must be a positive integer " * "(given $(min_samples_leaf))")

elseif min_samples_split < 2
    throw("min_samples_split must be at least 2 " * "(given $(min_samples_split))")
end
end
end

```

该函数主要的工作便是对参数之间的约束关系进行检查，如果不满足则通过 `throw()` 函数抛出异常，并上报错误的原因。



提示 在项目的整个实现中，该函数并非核心过程，是一个工程性的辅助函数。在这里，我们暂不关心那些约束条件，值得我们学习的是这种模块化的处理方式。为了确保程序的健壮性，确保程序执行的前置条件正确，专门设计一个函数对输入进行检查，不但能够将这个功能代码进行隔离以便于维护，也不会影响主线逻辑的清晰度，是非常好的编码习惯。

对输入参数检查后，`fit()` 函数利用 `let` 结构创建了初始的 `NodeMeta` 对象（根节点），并创建了 `Tree` 对象，绑定到了 `tree` 变量中。接着在对各种变量进行初始化之后，便是在 `while` 循环中反复地调用 `_split!()` 函数，执行决策树分裂的核心算法。在每一层循环中，会对 `_split!()` 处理后的 `NodeMeta` 结构进行分拆处理，实现的函数如下：

```

@inline function fork!(node :: NodeMeta)
    ind = node.split_at          # 新分支生成的分割点
    region = node.region         # 继承父节点信息
    features = node.features     # 继承父节点信息

    # 生成左节点: region更新, 深度增加一度
    node.l = NodeMeta(features, region[1:ind], node.depth + 1)
    # 生成右节点: region更新, 深度增加一度
    node.r = NodeMeta(features, region[ind+1:end], node.depth + 1)
end

```

该函数比较简单，主要是将 `node` 节点的一些信息传递到左右孩子节点中，同时为各分支提供不同范围的样本集 `region`。

至此，`treeclassifier` 模块中 `fit()` 函数的主体实现便介绍完了。接下来，便是实现核心过程的 `_split!()` 函数。因为该函数的实现依赖于决策树模型相关的各种理论和方法，因篇幅所限，这里不再继续详述。



在 <https://gitee.com/juliapro/DecisionTree.jl.git> 提供的源码中，笔者已经进行了详尽的注释。读者可以在熟悉决策树的基础上，对其进行深入的学习。

模型训练完之后，便可用其对测试数据进行预测。由于预测过程的实现较为简单，主要是 `apply_tree()` 函数的各个实现方法，本文也不再多做介绍。

17.3 随机森林算法的构建

在介绍 `DecisionTree.jl` 对随机森林的实现过程之前，我们先了解一下集成学习 (Ensemble Learning) 的概念。

集成学习是指将多个学习器进行组合集成，以共同完成对样本数据的学习目标。这其实是一种极为朴素的思维方式。通常而言，一个学习模型总会有自己独特的优缺点，在某个数据集上的表现总不会那么好，但如果通过某种方式将各个模型进行组合，便能够达到协同强化的效果。

常见的集成方式主要有提升法 (Boosting) 和套袋法 (Bagging) 两种，如下所示：

- 在 Boosting 方式中，多个学习器采用串行的连接，存在着很强的依赖关系。这是一种将弱学习器 (Weak Learner) 不断提升为强学习器的过程：首先基于初始的样本集训练出一个基学习器 (Base Learner)，然后根据该学习器的表现对样本进行调整，让分错的样本得到更多关注，接着再次对下一个基学习器进行训练，并不断重复此过程直至最终目标满足要求为止。最具代表性的 Boosting 方案是 AdaBoost，其较为容易理解的具体实现是累加型模型 (Additive Model)，即是将得到的各基学习器进行加权的线性组合，以达到最优的效果。
- 在 Bagging 方式中，不同的学习器相对独立，分别随机地从样本集总体中进行采样并进行独立学习。在各学习器完成训练后，Bagging 中对这些子模型进行组合的最简单方式便是在分类情况下采用投票法，而在回归情况下采用平均法。不过在这种方式中，如果样本子集差异较大，得到的学习器很可能也会出现很大的差异，导致组合时出现难度。为此，在为不同学习器进行采样时，尽量让各样本子集有一定的交叠。

当 Bagging 与决策树结合时，便是随机森林算法；当 Boosting 与决策树结合时便是提升树；若是决策树与 Gradient Boosting 集合，便得到了应用广泛的 GBDT 模型。因篇幅所限，本书不便展开，在接下来的内容中仅对 `DecisionTree.jl` 包中的随机森林算法做一下简单的介绍。

随机森林的基本结构在 `src/DecisionTree.jl` 脚本中定义为：

```
struct Ensemble
    trees::Vector{Node}
end
```

可见非常简单，是一个不可变复合类型，即 `trees` 成员后期不能再重新绑定到其他数组，但已绑定的 `Vector` 本身却是可以改变的。可以看出，实际上将森林表示为树结构 (`Node`)

的数组。

对随机森林训练的主要函数在 `src/classification/main.jl` 中实现, 具体代码为:

```
function build_forest(labels::Vector, features::Matrix, n_subfeatures=0, n_trees=10,
                    partial_sampling=0.7, max_depth=-1; rng=Base.GLOBAL_RNG)
    rng = mk_rng(rng)::AbstractRNG
    partial_sampling = partial_sampling > 1.0 ? 1.0 : partial_sampling
    Nlabels = length(labels)
    Nsamples = _int(partial_sampling * Nlabels)

    # 逐个对森林中的各个独立决策树模型进行训练
    forest = @distributed (vcat) for i in 1:n_trees
        inds = rand(rng, 1:Nlabels, Nsamples) # 随机采样
        build_tree(labels[inds], features[inds,:],
                  n_subfeatures, max_depth; rng=rng)
    end

    return Ensemble([forest;])
end
```

在进行一些必要的初始化之后, 利用 `for` 结构针对单个的决策树学习器进行随机采样, 然后利用 `build_tree()` 函数对单个学习器进行训练。在代码中, 使用 `@distributed` 宏进行修饰, 让各决策树学习器的训练过程能够并行地执行, 而将得到的结果绑定到 `forest` 变量中。此后, 以该变量为基础, 创建 `Ensemble` 对象并返回, 得到了训练好的随机森林模型。

在随机森林训练完成后, 便可使用 `apply_forest()` 函数对新样本进行预测。该函数的实现代码如下:

```
function apply_forest(forest::Ensemble, features::Vector)
    n_trees = length(forest)
    votes = Array{Any}(n_trees)

    # 逐一使用单个决策树学习器进行预测
    for i in 1:n_trees
        votes[i] = apply_tree(forest.trees[i], features)
    end

    # 根据类型选择不同的投票方法
    if typeof(votes[1]) <: Float64 # 类型断言
        return mean(votes)
    else
        return majority_vote(votes)
    end
end

function apply_forest(forest::Ensemble, features::Matrix)
    N = size(features,1)
    predictions = Array{Any}(N)

    # 对每个特征向量进行预测
    for i in 1:N
        predictions[i] = apply_forest(forest, features[i, :])
    end
end
```



```
# 采用点操作符统一对各特征向量的预测结果进行投票
if typeof(predictions[1]) <: Float64
    return Float64.(predictions)
else
    return predictions
end
end
```

其中有两种方法，前者适用于单个特征向量，后者适用于特征矩阵（即多个特征向量）。从代码中能够看出，后者针对每个特征向量调用了前者，然后采用点操作符对各特征向量的预测结果进行投票。

至此为止，我们已经介绍完了随机森林的主要实现，更多的细节可通过源码进行学习。至于提升树及回归实现限于篇幅也不再继续多做介绍，在读者自行学习的过程中，如遇到任何问题，都可以通过邮件等方式与笔者进行交流。

内置异常类型

在 Julia 内置的异常处理机制中，对已知的、常见的错误进行了定义，并使用特定的类型进行标识。这些类型均是 `Exception` 的子类型，列举如表所示：

异常类型	说明
<code>Base.ArgumentError</code>	调用一个函数时，提供的参数不能匹配该函数各实现方法
<code>Base.AssertionError</code>	使用 <code>assert</code> 命令对条件进行判断时，表达式的取值不为 <code>true</code> 值
<code>Core.BoundsError</code>	访问数组时，索引值超出了该数组的合理索引范围
<code>Base.DimensionMismatch</code>	访问数组时，维度数不匹配
<code>Core.DivideError</code>	整型除法中，分母为 0
<code>Core.DomainError</code>	函数或构造方法的参数超出了有效域
<code>Base.EOFError</code>	文件或流对象中没有更多的数据可读
<code>Core.ExceptionError</code>	这是一般的错误类型，可从其变量 <code>msg</code> 中获得错误细节
<code>Core.InexactError</code>	类型转化无法准确地进行
<code>Core.InterruptException</code>	进程被中断（可由 <code>CTRL+C</code> 引发）
<code>Base.KeyError</code>	对关联性容器（如 <code>Dict</code> ）或集合（ <code>Set</code> ）对象中不存在的元素进行访问而引发的错误
<code>Base.LoadError</code>	在通过 <code>include</code> 、 <code>require</code> 或 <code>using</code> 等命令访问文件时，发生加载错误
<code>Base.MethodError</code>	在给定的基本函数中无法找到匹配的实现方法，或者候选方法不唯一，导致歧义
<code>Core.OutOfMemoryError</code>	内存分配超出系统限制或超出垃圾回收的处理能力
<code>Core.ReadOnlyMemoryError</code>	访问只读的内存区域
<code>Core.OverflowError</code>	表达式结果超出给定类型的表达能力，引发了 <code>wraparound</code> 问题

(续)

异常类型	说 明
Base.ParseError	调用 parse 函数时, 无法解析成 Julia 有效的表达式
Core.StackOverflowError	函数调用过程导致了栈溢出, 常见的原因是出现了无限递归或死循环
Base.SystemError	系统级调用时发生错误 (错误信息保存于 errno 全局变量)
Core.TypeError	类型断言错误, 或以错误的参数类型调用了内置函数
Core.UndefRefError	对象或其成员元素未定义
Core.UndefVarError	当前可见域内的符号未定义
Base.InitError	模块初始化时 (调用其 __init__ 函数) 发生错误 (具体的错误信息可参见 .error 字段)

内置系统常量

Julia 内置系统常量见下表。

系 统 常 量	说 明
Core.nothing	类型 Void 的单例对象，用于无值可返回等情况。该常量在打印时不会显示任何内容，可转为空的 Nullable 值
Base.PROGRAM_FILE	命令行启动 Julia 时接收到的脚本文件路径，但通过 Julia 命令直接执行表达式时，例如 <code>julia -e <expr></code> ，会返回 nothing 对象；在 include 文件中，该变量指该文件的绝对路径，并保持不变，也可使用宏 <code>@__FILE__</code> 获得其所处脚本的绝对路径
Base.ARGS	包含通过命令行传递给 Julia 程序的参数列表，为字符串数组类型
Base.C_NULL	对应于 C 语言中的空指针常量
Base.VERSION	<p>VersionNumber 类型的常量，记录当前正使用的 Julia 版本号，包括主版本号 (major)、次版本号 (minor) 与补丁版本号 (patch)，还有预发布 (prerelease) 及构建 alpha 说明信息。例如：</p> <pre>julia> Base.VERSION v"1.0.0"</pre> <p>指当前 Julia 版本为 1.0.0</p> <p>VersionNumber 类型是一种特殊字符串，专门用于表达版本信息，例如：</p> <pre>julia> dump(v"0.2.1-rc1+win64") VersionNumber major: Int64 0 minor: Int64 2 patch: Int64 1 prerelease: Tuple{String} 1: String "rc1" build: Tuple{String} 1: String "win64"</pre> <p>其中，major 版本为 0，minor 版本为 2，patch 版本为 1，prerelease 信息为 rc1，及构建信息为 win64。一般来说，除了 major 版本号，其他都是可选的</p>

(续)

系 统 常 量	说 明
Base.VERSION	<p>VersionNumber 的对象是可以相互比较的, 例如:</p> <pre>julia> v"0.2" <= v"0.3-" true</pre> <p>其中, “-” 是 Julia 对标准规则的扩展, 指要小于包括预发布的任意 0.3 发布版。这种版本规则也用于 Julia 的包管理, 有兴趣的读者可查阅资料以了解更多信息</p>
Base.LOAD_PATH	一个字符串数组常量, 列出 require 函数、using 或 import 命令加载对象或代码时的搜索目录
Base.Sys.BINDIR	字符串常量, 指向 Julia 可执行主程序所在目录的绝对路径
Core.ANY	<p>在多态分发中相当于 Any 类型, 但会让编译器忽略其中的代码生成特例化 (code generation specialization), 其实际结构为:</p> <pre>TypeVar name: Symbol ANY lb: Core.TypeofBottom Union{} ub: Any</pre>
Base.Sys.CPU_THREADS	系统中可用逻辑内核 (包括 CPU 物理内核及超线程) 的数量。使用 Hwloc.jl 包能够获得关于 CPU 更多的信息, 包括内存、缓存、网络、物理内核等
Base.Sys.WORD_SIZE	当前机器运行环境的机器字节长度 (位数), 目前大多是 64 位
Base.Sys.KERNEL	Symbol 对象, 记录操作系统的内核名, 例如 Windows 10 中会返回 :NT 内容
Base.Sys.ARCH	Symbol 对象, 记录系统架构。在 64 位 PC 系统中, 一般为 :x86_64
Base.Sys.MACHINE	字符串对象, 记录 Julia 程序的构建信息。例如, 在 Windows 中, 其内容类似于 "x86_64-w64-mingw32"
Base.stdout	标准输出流全局变量
Base.stderr	标准错误流全局变量
Base.stdin	标准输入流全局变量
Base.ENV	一个 Base.EnvHash 类型的单例对象, 以字典结构存储了操作系统的环境变量
Base.ENDIAN_BOM	一个 UInt32 类型的数值, 记录机器的字节顺序标记值。对于低字节序 (Little-endian), 其值为 0x04030201, 而对于高字节序 (Big-endian), 其值为 0x01020304
Base.DL_LOAD_PATH	字符串数组, 记录着动态库的搜索路径
Base.Libdl.dlext	字符串类型, 记录当前平台动态库文件的扩展名, 例如 dll、dylib 或 so 等
Base.devnull	IO 子类型 Base.DevNull 的常量, 等效于 Windows 中的 NUL 或 Linux/Unix 中的 /dev/null 设备符, 用于对流进行重定向, 写入到其中的所有数据会被丢弃

字符串操作函数

Julia 的字符串操作函数如下表。

操 作	函 数
去除头尾空白或指定字符	<p>函数 <code>strip(s::AbstractString, [c::Chars])</code>，若提供参数 <code>c</code>，则会只移除 <code>c</code> 指定的字符。例如：</p> <pre>julia> strip("{3, 5}\n", ['{', '}', '\n']) "3, 5"</pre> <p>另外还有 <code>rstrip()</code> 和 <code>lstrip()</code> 函数分别用于去除右侧或左侧的字符，可参考官方库文档了解</p>
删除尾部字符	<p>函数 <code>chop(s::AbstractString; head::Integer = 0, tail::Integer = 1)</code>，移除头部 <code>head</code> 个及尾部 <code>tail</code> 个字符。例如：</p> <pre>julia> a = "March" "March" julia> chop(a) "Marc" julia> chop(a, head = 1, tail = 2) "ar" julia> chop(a, head = 5, tail = 5) ""</pre>
删除尾部单个换行符	<p>函数 <code>chomp(s::AbstractString)</code>，例如</p> <pre>julia> chomp("Hello\n") "Hello"</pre>
前缀是否为指定内容	<p>函数 <code>startswith(s::AbstractString, prefix::AbstractString)</code>，判断 <code>s</code> 的前缀是否是 <code>prefix</code>，是则返回 <code>true</code>，否则返回 <code>false</code>。例如：</p> <pre>julia> startswith("JuliaLang", "Julia") true</pre>

(续)

操 作	函 数
后缀是否为指定内容	<p>函数 <code>endswith(s::AbstractString, suffix::AbstractString)</code>, 判断 <code>s</code> 的后缀是否是 <code>suffix</code>, 是则返回 <code>true</code>, 否则返回 <code>false</code>。例如:</p> <pre>julia> endswith("Sunday", "day") true</pre>
字符全部大写	<p>函数 <code>uppercase(s::AbstractString)</code>, 将 <code>s</code> 中的所有字符转变转为大写。例如:</p> <pre>julia> uppercase("Julia") "JULIA"</pre>
字符全部小写	<p>函数 <code>Unicode.lowercase(s::AbstractString)</code>, 将 <code>s</code> 中的所有字符转变转为小写。例如:</p> <pre>julia> lowercase("STRINGS AND THINGS") "strings and things"</pre>
所有单词首字母均大写	<p>函数 <code>Unicode.titlecase(s::AbstractString)</code>, 例如:</p> <pre>julia> titlecase("the julia programming language") "The Julia Programming Language"</pre>
首字母大写	<p>函数 <code>Unicode.uppercasefirst(s::AbstractString)</code>, 仅将字符串 <code>s</code> 中首字符大写, 例如:</p> <pre>julia> uppercasefirst("python") "Python"</pre>
首字母小写	<p>函数 <code>Unicode.lowercasefirst(s::AbstractString)</code>, 仅将字符串 <code>s</code> 中首字符小写, 例如:</p> <pre>julia> lowercasefirst("Julia") "julia"</pre>
字符是否为文字	<p>函数 <code>Unicode.isletter(c:: AbstractChar)</code>, 如果 <code>Unicode</code> 分类是 <code>Letter</code>, 则结果是 <code>true</code></p>
字符是否为数字	<p>函数 <code>Unicode.isnumeric(c:: AbstractChar)</code>, <code>Unicode</code> 类别为 <code>Number</code> 的字符 (类码首字母为 <code>N</code>)。例如:</p> <pre>julia> isnumeric('%') true julia> isnumeric('@') true julia> isnumeric('9') true julia> isnumeric('α') false</pre> <p>该函数会考虑 <code>Unicode</code> 中那些被标识为数字类型各种字符, 如果仅区分 <code>0~9</code> 的十进制数字, 可使用 <code>isdigit()</code> 函数</p>
是否字符在 ASCII 中, 或字符串所有字符均在其中	<p>函数 <code>isascii(c::Union{AbstractChar, AbstractString})</code>, 是则返回 <code>true</code></p>

(续)

操 作	函 数
是否为控制字符	函数 <code>Unicode.iscntrl(c:: AbstractChar)</code> , 控制字符是 Unicode 的 Latin-1 子集中的不可打印字符
是否为 0~9 中某一个	函数 <code>Unicode.isdigit(c:: AbstractChar)</code> , 判断是否是十进制中的数字字符
字符是否可打印 (包括空格)	函数 <code>Unicode.isprint(c:: AbstractChar)</code> , 不包括控制字符
字符是否为标点符号	函数 <code>Unicode.ispunct(c:: AbstractChar)</code> , Unicode 类别为 Punctuation
是否为空白字符	函数 <code>Unicode.isspace(c:: AbstractChar)</code> , 包括 ASCII 字符的 '\t'、'\n'、'\v'、'\f'、'\r' 和 ' ', Latin-1 字符 U+0085, Unicode 类别为 Zs 的字符
字符是否小写	函数 <code>Unicode.islowercase(c:: AbstractChar)</code> , 任何 Unicode 类别为 Ll、Letter: Lowercase 中的字符
字符是否大写	函数 <code>Unicode.isuppercase(c:: AbstractChar)</code> , 任何 Unicode 类别为 Lu、Letter: Uppercase 或 Lt、Letter: Titlecase 中的字符

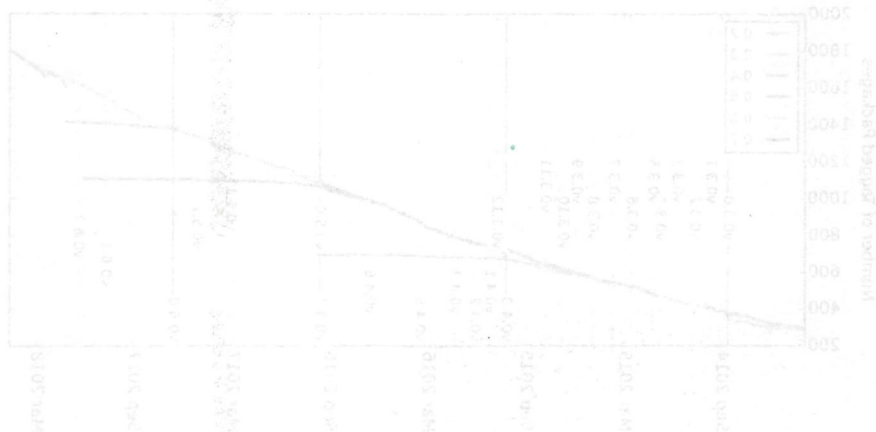


图 D-1 Julia 温度转换

常用包简介

第三方开发的 Julia 库（包），是 Julia 生态中极为重要的一部分。目前 Julia 包虽然没有 Python 那样丰富，但却极为快速地发展着。官方提供了包数量的统计趋势，如图 D-1 所示。

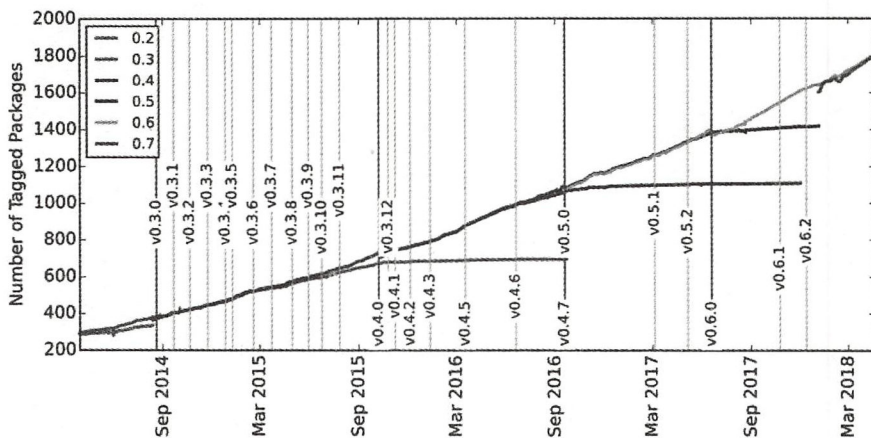


图 D-1 Julia 包发展趋势

从图 D-1 可见，Julia 包的发展非常之快。相信在不久的将来，其丰富性及多样性能够覆盖开发中的各种场景，并能够快速渗透到科学计算等领域。读者可在 pkg.julialang.org 或者网站 juliaobserver.com 中的近两千个包中寻找适合自己的包。

下面我们将列举一些常用包，关于它们的详细使用方法，请参阅官方文档。

1. Documenter.jl

这是一个由官方维护的包，专门用于将 Julia 代码中的注释系统（Docstrings）及其中的 Markdown 文件生成完整的说明文档。

该包提供的文档生成器具备以下功能特点：

- ❑ 支持 Markdown 语法。
- ❑ 开箱即用 (Minimal Configuration)。
- ❑ 会对注释中标注的代码执行测试语句 (Doctest)。
- ❑ 支持文档章节等方面的交叉引用。
- ❑ 支持 LaTeX 语法。
- ❑ 检查文档字符系统缺失与交叉引用错误。
- ❑ 自动生成目录与文档索引。
- ❑ 使用 Git 自动构建与部署文档，并能推送到 GitHub 的文档库中。

包 `Documenter.jl` 是已经注册的包，安装很简单，只需执行 `Pkg.add("Documenter")` 命令即可。该包会将代码库的 Markdown 文件及代码中以 `docstrings` 方式注释的文档系统解析合并为单一的内联 (inter-linked) 文档。

2. DataFrames.jl

在科学计算中往往需要综合多种库的优势，以构造强大的应用与分析系统，一个通用的数据结构便成为整合这些不同库的纽带，能够为跨库调用带来极大的便利，更可为语言的快速发展奠定基础。

在 Python 语言中，有一个专门用于处理行列式或表格式 (tabular) 数据的结构，名为 `DataFrame`，是科学计算库 `Pandas` 的重要组成部分，已成为很多 Python 第三方包支持的基本数据操作框架。

与之类似，Julia 语言的 `DataFrames` 库，也意图建立这样一个基础的数据结构。通过 `Pkg.add("DataFrames")` 命令可安装该包，此后使用 `using DataFrames` 即可将该包导入到开发者的名字空间，使用 `DataFrames` 中的各种功能。

`DataFrame` 类型能够表达表格式数据，并支持数据中存在缺失值。其内部是列存储的，每列是一个向量结构。在构建 `DataFrame` 对象时，可以同时通过参数提供每列的名称，例如：

```
julia> using DataFrames
```

```
julia> df = DataFrame(A = 1:4, B = ["M", "F", "F", "M"])
```

```
4×2 DataFrames.DataFrame
```

Row	A	B
1	1	M
2	2	F
3	3	F
4	4	M

在构造方法中，参数名即是列名，而参数值即是对应列的取值。例中的 `DataFrame` 对象有两列：第一列名为 `A`，取 `1~4` 的整型；第二列名为 `B`，值是字符串类型；而列 `Row` 是默认添加的行编号。

我们可以通过一些函数查看 DataFrame 对象的属性：

```
julia> size(df)                # 取得DataFrame对象的行与列数
(4, 2)
```

```
julia> size(df,1)              # 取得对象的行数
4
```

```
julia> size(df,2)              # 取得对象的列数
2
```

也可以通过 describe() 函数取得基本统计信息：

```
julia> describe(df)
A                                # 列名
Summary Stats:
Mean:      2.500000              # 均值
Minimum:   1.000000              # 最小值
1st Quartile: 1.750000          # 分位数
Median:    2.500000              # 中值
3rd Quartile: 3.250000          # 分位数
Maximum:   4.000000              # 最大值
Length:    4                     # 行数
Type:      Int64                 # 数据类型
```

```
B
Summary Stats:
Length:    4                     # 行数
Type:      String                 # 数据类型
Number Unique: 2                 # 唯一值（去重后）的数量
```

当然，我们可以通过列名提取出某列的向量：

```
julia> df[:A]
4-element Array{Int64,1}:
 1
 2
 3
 4
```

```
julia> df[:B]
4-element Array{String,1}:
 "M"
 "F"
 "F"
 "M"
```

需要注意的是，代码中使用列名时，需使用 Symbol 类型表达。

3. CSV.jl

简单的数据交换与存储，可采用 CSV 格式，这是一种常用的表格式数据，而包 CSV.jl 专为 CSV 文件的读写提供支持。

通过 Pkg.add("CSV") 后，即可使用该包。该包非常简单，基本上就两个操作——read() 和 write() 函数。我们下面仅以 read() 函数为例，介绍该包的使用。

函数 `read()` 会从指定的文件中加载 csv 格式的数据，并存储到指定的数据结构中，默认的结构为 `DataFrame` 类型，其原型为：

```
CSV.read(fullpath::Union{AbstractString,IO}, sink::Type{T}=DataFrame, args...; kwargs...)
CSV.read(fullpath::Union{AbstractString,IO}, sink::Data.Sink; kwargs...)
```

其中，`fullpath` 指定 csv 文件的绝对路径，或者是已经打开的某个 IO 对象。其中的 `Data.Sink` 是第三方包 `DataStreams.jl` 提供的类型，可参阅相关文档，这里以 `Data-Frame` 类型为例。

假设有一个 csv 文件，其内容如表 D-1 所示，各列以英文逗号分隔。

表 D-1 csv 文件示例内容

c1	c2	c3
1	7.2	abc
12	8.4	def
13	9.3	ghi

通过 CSV 包读取，便可将该 csv 文件的内容加载到一个 `DataFrame` 对象中，即：

```
julia> using CSV
julia> data = CSV.read("d:/test.csv")
3×3 DataFrames.DataFrame
 | Row | c1 | c2 | c3 |
 |----|---|---|---|
 | 1   | 11 | 7.2 | abc |
 | 2   | 12 | 8.4 | def |
 | 3   | 13 | 9.3 | ghi |
```

之后，便可基于 `DataFrames` 包的功能进行各种后续操作，例如，利用 `describe()` 查看基本情况：

```
julia> describe(data)
c1
Summary Stats:
Mean:          12.000000
Minimum:       11.000000
1st Quartile:  11.500000
Median:        12.000000
3rd Quartile:  12.500000
Maximum:       13.000000
Length:        3
Type:           Union{Int64, Missings.Missing}
Number Missing: 0
% Missing:      0.000000

c2
Summary Stats:
Mean:          8.300000
Minimum:       7.200000
1st Quartile:  7.800000
Median:        8.400000
3rd Quartile:  8.850000
Maximum:       9.300000
Length:        3
Type:           Union{Float64, Missings.Missing}
Number Missing: 0
% Missing:      0.000000
```



```

c3
Summary Stats:
Length:      3
Type:        Union{Missings.Missing, String}
Number Unique: 3
Number Missing: 0
% Missing:   0.000000

```

可见, CSV 能够自动识别出每列的数据类型。

假设上例中的数据有一个缺失值, 并以 N 字符表示, 如表 D-2 所示。

在读取数据时, 设置 `missingstring` 参数^①为 N, 告知 `read()` 函数有缺失值:

表 D-2 有缺失值的 csv 文件示例

c1	c2	c3
11	7.2	abc
12	N	def
13	9.3	ghi

```
julia> data = CSV.read("d:/test.csv"; missingstring="N")
```

```
3×3 DataFrames.DataFrame
```

Row	c1	c2	c3
1	11	7.2	abc
2	12	missing	def
3	13	9.3	ghi

可见, 在获得的 `DataFrame` 中, 列 `c2` 中的缺失值得到了正确的识别。

另外, 通过 `read()` 函数的键值参数还可以得知 csv 的其他各种格式情况, 以便在加载数据时能够成功处理。下面给出该函数的其他一些使用示例:

```

# 读取以空白符为分隔符的文件
CSV.read(file; delim=' ')

```

```

# 自定义列名, 适用于文件中没有提供表头的csv文件, 但需要列名的数量与文件的列数一致
CSV.read(file; header=["col1", "col2", "col3"])

```

```

# 自定义每列的数据类型, 但必须与实际的数据匹配
CSV.read(file; types=[Int, Int, Float64])

```

```

# 自定义读取的文件行数, 有助于提高解析速度, 并可用于只读取所需的有限头部数据
CSV.read(file; rows=10000)

```

4. JSON.jl

在数据交换中, 尤其是网络数据中, Json 格式的使用非常频繁, Julia 同样提供了一个简洁的库, 用于处理这类格式的数据。

假设有个名为 `s` 的字符串变量, 存储了 Json 数据, 如下所示:

```

{
  "a_number": 5.0,
  "an_array": ["string", 9],
  "inner": {
    "value": 3
  }
}

```

^① 旧版 CSV 包中使用键值参数 `null` 指定缺失值的标识字符串, 但新版中已弃用, 被 `missingstring` 代替。

在安装 JSON 包后，可以尝试将其解析到数据结构，语句非常简单：

```
julia> s = "{\"a_number\" : 5.0, \"an_array\" : [\"string\", 9], \"inner\": {\"value\": 3}}";

julia> j = JSON.parse(s)
Dict{String,Any} with 3 entries:
  "an_array" => Any["string", 9]
  "a_number" => 5.0
  "inner"    => Dict{String,Any} {Pair{String,Any} ("value", 3)}
```

函数 `parse()` 会将 `s` 的内容分析出来，放在一个嵌套的 `Dict` 对象中，并会自动进行数据类型识别，然后就可以通过关键字进行层层提取了，代码如下：

```
julia> j["an_array"]
2-element Array{Any,1}:
 "string"
 9

julia> j["inner"]
Dict{String,Any} with 1 entry:
 "value" => 3

julia> j["inner"]["value"]
3
```

反之，也可以将对象结构转换到 `Json` 字串，便于存储或传输：

```
julia> JSON.json(j)
"{\"an_array\":[\"string\",9],\"a_number\":5.0,\"inner\":{\"value\":3}}"
```

5. Taro.jl

这是一个基于 Apache Tika 等库开发的强大库，用于从 Word、Excel 或 PDF 文件中提取内容，其有以下几个功能点：

- ☐ 能够从 Word、Excel、PDF 文件中提取原始的文本数据。
- ☐ 从 Excel 文件中将数据提取到 `DataFrame` 结构中。
- ☐ 以 Julia 语言读取 Excel 文件的 API 接口函数。
- ☐ 将采用 XSL-FO 布局的文件转换为 PDF 文件。

6. Gadfly.jl

包 Gadfly 是 Julia 语言开发的绘图与可视化库，由 Daniel C. Jones 设计，现在由社区维护，其特点包括：

- ☐ 相当于 SVG、PNG、Postscript 与 PDF 的图形渲染质量。
- ☐ 直观的、一致性的绘图接口设计。
- ☐ 可在 IJulia 中使用。
- ☐ 能够使用 `DataFrames.jl` 数据结构。
- ☐ 支持大量的绘图类型。

Gadfly 采用了一种称为“图形语法”的标准进行架构的设计，其在绘图命令提交后，

会遵循内部的流水线进行渲染，得到最终的结果。

渲染流水线会将绘图命令转为一个整体性的图形场景，包括坐标轴、颜色等导引元素，分层的点、线等图形元素。每一层包括数据源、图形元素（简称图元）、数据与图元的关联映射、应用于层次数据统计等部分。所有的层共享如下元素：坐标系（如笛卡尔坐标系、极坐标系等）、坐标轴的缩放（如重对数、半对数等），图形场景统计及导引元素等。

一套完整的绘制命令需要描述清楚这些共享的元素及各层的规格。完整的渲染进程如图 D-2 所示。

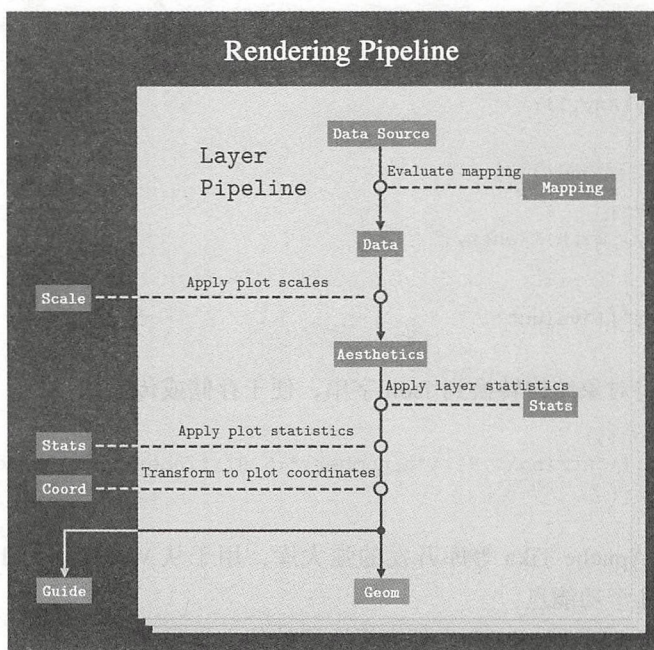
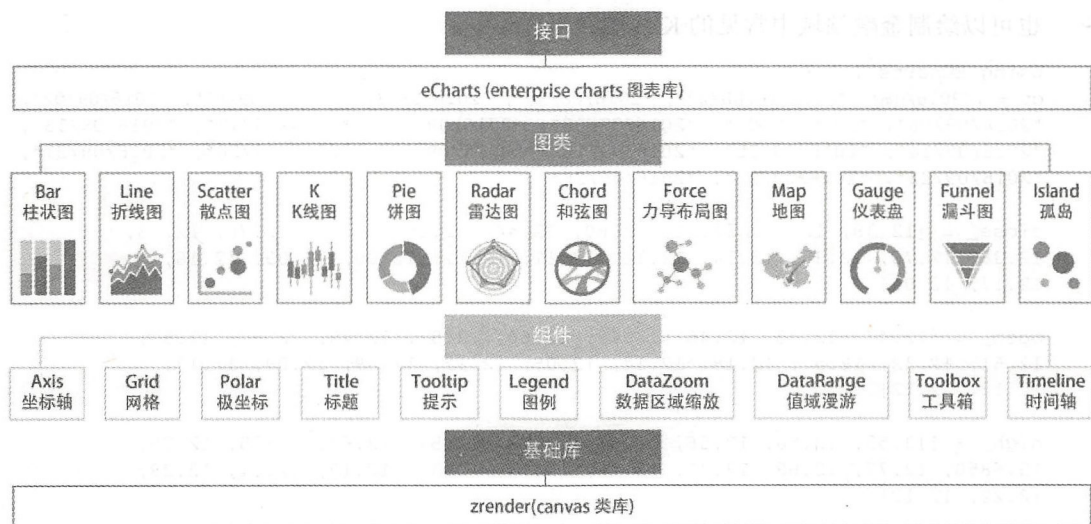


图 D-2 Gadfly 渲染流水线框架

7. ECharts.jl

ECharts 是一款由百度前端技术部开发的，基于 JavaScript 的数据可视化图表库，提供直观、生动、可交互、可个性化定制的数据可视化图表，如图 D-3 所示。提供大量常用的数据可视化图表，底层基于 ZRender（一个全新的轻量级 canvas 类库），创建了坐标系、图例、提示、工具箱等基础组件，并在此上构建出折线图（区域图）、柱状图（条状图）、散点图（气泡图）、饼图（环形图）、K 线图、地图、力导向布局图以及和弦图，同时支持任意维度的堆积和多图表混合展现。

包 ECharts.jl 基于 Julia 语言对 JavaScript 的 ECharts 4 进行了封装，从而能在 Julia 中使用这个强大的图表库。在 `Pkg.add("ECharts")` 之后，即可通过 `using ECharts` 使用该包。

图 D-3 Echarts 框架图^①

例如，我们要绘制一个柱状图：

```
using ECharts
x = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"]
y = [11, 11, 15, 13, 12, 13, 10]
bm = bar(x, hcat(0.95 .* y, 1.25 .* y, y))
```

可获得柱状图如图 D-4 所示。

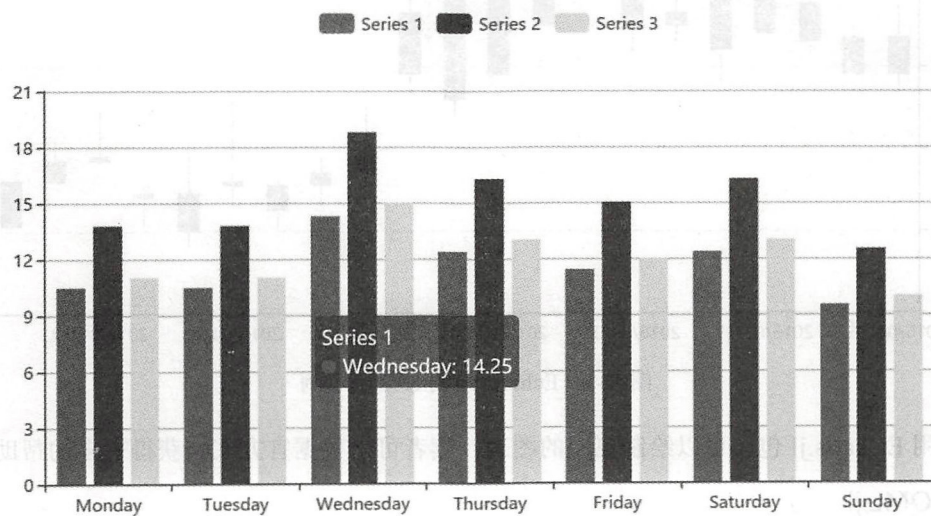


图 D-4 Echarts 绘制柱状图示例

^① 图片来自 <https://www.oschina.net/p/echarts>。

也可以绘制金融领域中常见的 K 线图：

```
using ECharts
dt = ["2016/08/26", "2016/08/29", "2016/08/30", "2016/08/31", "2016/09/01", "2016/09/02",
      "2016/09/06", "2016/09/07", "2016/09/08", "2016/09/09", "2016/09/12", "2016/09/13",
      "2016/09/14", "2016/09/15", "2016/09/16", "2016/09/19", "2016/09/20", "2016/09/21",
      "2016/09/22", "2016/09/23", "2016/09/26"]

close_ = [12.38, 12.47, 12.55, 12.60, 12.44, 12.50, 12.67, 12.70, 12.73,
          12.38, 12.70, 12.38, 12.14, 12.11, 12.11, 12.11, 12.00, 12.09, 12.18,
          12.17, 12.01]

open_ = [12.47, 12.38, 12.46, 12.48, 12.66, 12.53, 12.49, 12.62, 12.50,
          12.61, 12.32, 12.53, 12.18, 12.14, 12.05, 12.12, 12.09, 12.08, 12.17,
          12.12, 12.12]

high_ = [12.55, 12.50, 12.56, 12.61, 12.72, 12.57, 12.67, 12.75, 12.75,
          12.6850, 12.77, 12.68, 12.31, 12.18, 12.13, 12.33, 12.19, 12.11, 12.29,
          12.22, 12.12]

low_ = [12.34, 12.38, 12.43, 12.48, 12.35, 12.46, 12.43, 12.62, 12.50,
          12.38, 12.28, 12.33, 12.11, 12.06, 12.01, 12.0586, 11.96, 12.01, 12.16,
          12.12, 12.00]

c = candlestick(dt, open_, close_, low_, high_)
```

执行后获得的 K 线图如图 D-5 所示。

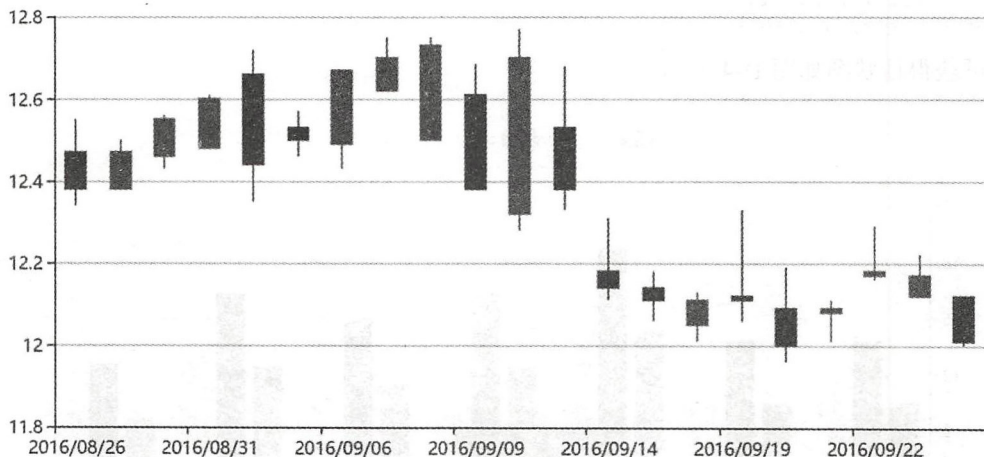


图 D-5 Echarts 绘制 K 线图示例

利用 ECharts.jl 包，可以绘制更多的图表，读者可以根据官方文档获得更多的帮助。

8. QML.jl

在图形用户界面 (Graphical User Interface, GUI) 方面，QT 是基于 C++ 开发的跨平台框架，支持包括 Windows、Linux、Android、WP、IOS 等多种系统平台，是笔者非常欣赏的用 C++ 语言开发的一个 GUI 套件，其简洁明了的接口和框架，使用起来非常方便，而且

界面控制也非常容易。借助 QT 官方提供的 QT Creator 工具，能够快速地设计出适用的交互界面。而且，QT 不限于开发 GUI，更适用于常见场景的应用开发，包括控制台工具、服务器、网络通讯等。该套件提供社区版，开发者只要遵循其协议，可免费使用。

在界面实现时，除了使用 C++ 语言设计交互逻辑外，还可以使用一种类似于 JavaScript 语言的名为 QML 的脚本语言进行界面设计，而在 Julia 中，包 QML.jl 即是通过 Cxx-Wrap.jl 包提供了 Qt5 QML 的编程支持。

在采用 Julia 进行 GUI 编程时，还可以参看 PySide.jl 包、Qwt.jl 包及 Gtk.jl 包。

9. JDBC.jl

数据库的操作在 Julia 中也非常方便，其中的 JDBC.jl 包提供了数据库访问的支持。

该包基于 JavaCall.jl 包的功能，通过 Java 的接口访问 JDBC 驱动。提供的 API 包括两种核心组件，一种是访问 JDBC 的直接接口，另一种是支持 DataStreams.jl 的 Julia 接口。

在使用该包前，需要安装好对应的 JDBC，并配置了正确的路径，然后需先对 JVM 进行初始化，如下：

```
using JDBC
JDBC.usedriver("/home/me/derby/derby.jar") # 路径指向驱动包
JDBC.init() # 或JavaCall.init()
```

需要注意的是，在使用该包后，需要显式地进行资源释放：

```
JDBC.destroy() # 或JavaCall.destroy()
```

另外，在 <https://github.com/JuliaDatabases> 中列出了更多关于数据库操作的 Julia 包，例如 SQLite.jl 包、ODBC.jl 包、PostgreSQL.jl 包、MySQL.jl 包等。

10. Distributions.jl

概率分布在科学计算中常常遇到，例如样本分析、概率表达等。包 Distributions 在概率分布方面提供了很好的支持。其功能特点包括：

- ❑ 矩（包括均值、方差、偏度、峰度等）的计算，还有熵及其他属性。
- ❑ 概率密度函数及其对数表达。
- ❑ 矩生成函数与特征函数。
- ❑ 样本采样或总体采样。
- ❑ 最大相似度估计。

该包同时还支持多元统计分布、混合模型、假设检验及分布拟合等计算。

11. Distances.jl

这又是一个强大的包，提供了各种相似性度量函数的支持。所谓相似性度量，是在模式识别与机器学习中必然会用到的基础模块之一，用于评估两个特征向量的距离，即相似性程度。采用何种距离函数，不论是在聚类还是在分类中，都需要认真考虑并慎重选择。

该包在相似度计算的实现中，进行了优化，相比于粗糙地使用迭代的方式实现距离评估，有着很大的性能优势。有兴趣的读者，可在官方包中找到对应的 Benchmark 对比数据。

表 D-3 列出其支持的所有距离函数，在开发中可自行选择。关于这些距离函数的具体定义，也请读者能够自行查找些资料，仔细研究。

表 D-3 Distances 支持的距离算法

中 文 名	英 文 名	中 文 名	英 文 名
欧式距离	Euclidean Distance	瑞利散度	Rényi divergence
平方欧式距离	Squared Euclidean Distance	JS 散度	Jensen-Shannon divergence
街区距离	Cityblock Distance	马氏距离	Mahalanobis distance
杰卡德距离	Jaccard Distance	平方马氏距离	Squared Mahalanobis distance
Rogers-Tanimoto 距离	Rogers-Tanimoto Distance	巴氏距离	Bhattacharyya distance
切比雪夫距离	Chebyshev Distance	黑林格距离	Hellinger distance
闵可夫斯基距离	Minkowski Distance	半正矢距离	Haversine distance
海明距离	Hamming Distance	平均绝对偏差	Mean absolute deviation
余弦相似度	Cosine Distance	均方差	Mean squared deviation
相关系数	Correlation distance	均方根偏差	Root mean squared deviation
卡方距离	Chi-square distance	标准化的均方根偏差	Normalized root mean squared deviation
KL 散度	Kullback-Leibler divergence	布雷柯蒂斯相异度	Bray-Curtis dissimilarity
标准化 KL 散度	Generalized Kullback-Leibler divergence		

另外，这些相似性度量在包中的具体实现还会有很多的变体，使我们有更为丰富的选择。值得注意的是，在使用这些距离函数时，最好能很清楚这些函数的定义方式，最为重要的一点是，这些距离函数的计算结果，是越大越表示相似，还是越小越表示相似。

12. TensorFlow.jl

该包是对 Google 开发的深度学习库 TensorFlow 的封装，只需执行如下代码即可安装此包：

```
Pkg.add("TensorFlow")
```

若要在 Linux 上支持 GPU 加速，可设置环境变量 TF_USE_GPU 为 1，然后重新编译，即：

```
ENV["TF_USE_GPU"] = "1"
Pkg.build("TensorFlow")
```

该包现已支持的功能包括：

- ❑ 所有基本的一元及二元数学函数或运算符。
- ❑ 最常用的神经网络操作，包括卷积、GRU 版的循环神经网络及 dropout 等。
- ❑ 神经网络训练模块，例如 AdamOptimizer 等。
- ❑ 基本的图像加载及缩放操作。

- ❑ 控制逻辑操作。
- ❑ PyBoard 图可视化。

13. Mocha.jl

包 Mocha 是 Julia 语言实现的深度学习框架，类似于 C++ 中的 Caffe 库。Mocha 高效地实现了常见的随机梯度优化和通用层，通过无监督的自编码器的学习方式，可用于训练深度神经网络（Deep Neural Networks, DNN）和卷积神经网络（Convolutional Neural Networks, CNN）。若要安装 Mocha 库，只需在 Julia 的 REPL 中运行如下命令：

```
Pkg.add("Mocha")
```

Mocha 的特点包括：

- ❑ 模块化的架构。Mocha 有着清晰的组织结构，而且各个组件间是低耦合的（isolated）。这些模块包括网络层、激励函数、优化过程、正则化函数、初始化函数等。内置的组件已足够用于典型的 DNN 或 CNN 应用，开发者更可在其基础上进行类型扩展。
- ❑ 高层接口。借助 Julia 语言强大的表达能力及迅速发展的开源生态，在 Mocha 中使用 DNN 等功能会很容易、自然。开发者可通过提供的图像分类案例进行方便的学习。
- ❑ 可移植及高性能。Mocha 因为基于 Julia 语言，故可运行于任何支持 Julia 语言的平台上，而且这种移植性对于开发者来说是透明的。利用 Julia 语言 JIT 编译器与各种内置的性能函数，可以方便、高效地进行算法的原型设计，并能够直接获得性能评估。而且，在 C++ 编译器可用时，原生扩展后端打开后能够比纯粹的 Julia 后台快 2~3 倍。
- ❑ Mocha 提供的 GPU 后台使用的是 NVidia 的 cuDNN 库、cuBLAS 库及定制的 CUDA 内核，提供了高性能的计算能力。采用 GPU 加速后，在现代 GPU 设备中，特别是在大型模型中，能够获得 20~30 倍的速度提升。
- ❑ 兼容性。借助专用于科学数据存储的 HDF5 格式，在 Mocha 用其存储数据集及模型的快照，能够很容易地与 Matlab、Python（Numpy）及其他已存在的计算库进行数据交换。Mocha 还提供了从 Caffe 中导入训练模型快照的工具，开发者可以充分利用不同库的优势。
- ❑ 可靠性（Correctness）。Mocha 中的每个组件都经过了充分的测试，保证了其正确性与可靠性。
- ❑ 开源开放。Mocha 采用的 MIT（Expat）许可证，其源代码是开源的，开发者可以方便地进行学习，并可自行扩展。

后 记

经过本书的学习，我们能够深刻体会到 Julia 语言在数值计算、类型系统、函数多态、高维数组、网络通信、并行计算以及混合编程方面的强大特性与优势。借助其灵活、轻便的语法机制，我们能够快速开发出简洁、高效的程序代码；更可基于其庞大、完善的包，在活跃的开源社区支持下，为各种场景提供性能优异的应用。

Julia 语言博采众长，语法灵活多样，不但支持面向对象范式，还支持泛型编程、函数式编程以及元编程等方式。其中涉及的概念可谓庞杂繁多，若要线性地阐述是很有挑战性的。本书采用了引导、发现的阐述方式，通过大量精选的示例代码，边探索求证边总结，一步步地引出语言的各种原理、语法或规则，希望通过这种方法能够以最直接的方式阐明每一个知识点。

在通过本书学习 Julia 时，建议读者在必要时回顾之前学习的内容，并能够主动查阅资料，了解本书未尽的内容。也希望读者能够在 Julia 环境中动手操作，多多实践，相信能更快地学会 Julia 这门独具特色的语言，也能对其有更为深刻的认识。

为了能够清晰、透彻地阐明每个原理与机制，在撰写本书期间，笔者反复斟酌用词，以求简洁明了，不让内容变得繁复冗长，而且尽量做到在关键处能够提纲挈领地阐明语言规则，以期让本书成为读者实践 Julia 的行动指南。

本书在撰写过程中，希望能够全面地介绍 Julia 语言的方方面面，内容一再扩充，但至截稿时仍觉无法深入地阐述其各种特点，只能点到为止，以期读者能够通过官网或社区资料进行更多的学习。

本书仍有很多不尽人意之处，希望能在与读者的交流中不断地改进和完善。笔者非常感谢机械工业出版社及吴怡编辑提供了撰写本书的机会，也正因为出版社各位编辑的大力支持，本书在经历数月的不断修改后才能够顺利出版。

作者简介

魏坤

上海交通大学控制理论与控制工程专业人工智能研究方向博士毕业，曾就职于阿里巴巴、大众点评，任大数据挖掘与机器学习专家。他一直醉心研究Julia编程语言与应用，分享了大量广受好评的技术文章。

Julia语言程序设计

在互联网极速发展的时代，人工智能和机器学习越来越多地渗透到生活应用中，顺应这个趋势，Julia语言理所当然会蓬勃发展，魏坤博士这本书恰逢其时地给大家介绍了这门“非主流”的语言。本书结构清晰，例子翔实，不管是新手还是老鸟，都会受益匪浅！我会毫不犹豫地推荐本书推荐给每一位遇到的工程师和专家。

—— 王俊卿 上海开点信息科技有限公司创始人兼CEO

与魏坤认识十多年了，这本书仍秉承了他一贯严谨的风格，每个要点都力求准确、深入、全面，如果有学生想学习Julia语言，我会极力推荐这本书。

—— 陈曦 华东师范大学教授、博导

Julia是一个面向科学计算的高性能动态编程语言。本书深入浅出地对Julia语言的语法特性及其应用进行了全面的阐述，严谨清晰，通俗易懂，对编程、科学计算及数据分析感兴趣的读者都能从本书获益。

—— 柳畅 上海方谷信息技术有限公司CTO

本书中的示例代码所在网址为 <https://gitee.com/juliaprog>。



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/程序设计

ISBN 978-7-111-60757-1



9 787111 607571 >

定价: 99.00元